PHAR LAP

386|DOS-EXTENDER REFERENCE MANUAL

386|DOS-EXTENDER SDK

386 | DOS-Extender Reference Manual

Phar Lap Software, Inc.
60 Aberdeen Avenue, Cambridge, MA 02138
(617) 661-1510, FAX (617) 876-2972
dox@pharlap.com
tech-support@pharlap.com

First edition: October 1986.
Second edition: May 1988.
Third edition: January 1991.

Table of Contents

Figures and Tables

Preface

Welcome to 386 | DOS-Extender, the premier 80386/80486 protected-mode runtime environment for executing your 32-bit application programs under MS-DOS or PC-DOS.

This manual provides reference information for the protected-mode environment seen by application programs under 386 | DOS-Extender. We recommend that you read Chapter 3, which provides an overview of the application environment, and the first two sections of Chapter 2, which describe building a protected-mode program. Browse through the rest of the manual as needed to obtain specific information for writing your application. Throughout the manual there are references to MS-DOS, 386 | DOS-Extender, and BIOS system calls; these are documented in Appendices A, B, and C respectively.

This manual is not a tutorial. It does **not** teach how to program for the MS-DOS environment. Familiarity with 8086-family programming and with MS-DOS is assumed. The differences between programming the 80386 and previous members of the 8086 family are described in the *386 | ASM Reference Manual* and in References 5 and 7 listed below. For a discussion of programming for the MS-DOS environment, please see Reference 3.

References

1. Intel Corporation. *386™ DX Programmer's Reference Manual*, Number 230985, 1990.

2. Intel Corporation. *80387 Programmer's Reference Manual*, Number 231917, 1987.

3. Duncan, Ray. *Advanced MS-DOS Programming*. Redmond, Washington: Microsoft Press, 1988.

4. Duncan, Ray, et al. *Extending DOS*. Reading, Mass.: Addison-Wesley Publishing Company, 1990.

5. Morse, Stephen P., Eric J. Isaacson, and Douglas J. Albert. *The 80386/387 Architecture.* New York: John Wiley & Sons, Inc., 1987.

6. Intel Corporation. *i486TM Microprocessor Programmer's Reference Manual*, Number 240486, 1990.

7. Turley, James L. *Advanced 80386 Programming Techniques*. Berkeley, California: McGraw-Hill, Inc., 1988.

8. Nelson, Ross P. *The 80386 Book.* Redmond, Washington: Microsoft Press, 1988.

9. Cyrix Corporation. *EMC87 Reference Manual,* Order number L2001-003.

10. Weitek Corporation. *Abacus Software Designer's Guide.*

11. Schulman, Andrew, et al. *Undocumented DOS*. Reading, Mass.: Addison-Wesley Publishing Company, 1990.

12. The DPMI Committee. *DOS Protected Mode Interface (DPMI) Specification, Version 0.9*, Intel order number 240743-001, May 15, 1990.

Manual Conventions

This manual relies on certain conventions to convey types of information. On the following pages, these are the conventions:

`Courier`	indicates command line, switch syntax and examples; this typeface was chosen for its close resemblance to screen display and to differentiate actual command lines from documentation.
`italics`	indicate items that must have a user-entered name or value entered in place of the items in italics.
nnn	a decimal (base 10) number.
*nnn*h	a hexadecimal (base 16) number.

Introduction

386 | DOS-Extender provides an 80386/80486 protected-mode runtime environment for 32-bit application programs running under MS-DOS or PC-DOS. Programs that run in protected mode can directly access all memory present in the machine, including memory above one MB.

386 | DOS-Extender functions as a layer between MS-DOS and the application program. The application makes DOS and BIOS system calls in the standard way, using software interrupts. 386 | DOS-Extender intercepts these system calls and passes them through to MS-DOS in the real mode of the 80386. It automatically takes care of moving data between the real- and protected-mode address space, so the full power of MS-DOS is available regardless of a program's size or how it organizes memory. (Please see Figure 1-1.) 386 | DOS-Extender turns MS-DOS into a true protected-mode operating system.

Versions of 386 | DOS-Extender

There are two versions of 386 | DOS-Extender. The development version is included with the Phar Lap 386 | DOS-Extender Software Development Kit and is distributed as a file named RUN386.EXE. The runtime version is provided when a runtime license is purchased to permit redistribution of 386 | DOS-Extender when bound to an application, and is distributed as a file named RUN386B.EXE. From the point of view of the application program, there is no difference between the two versions. The only difference is in the way each program is run from the MS-DOS command prompt, as described in Chapter 2.

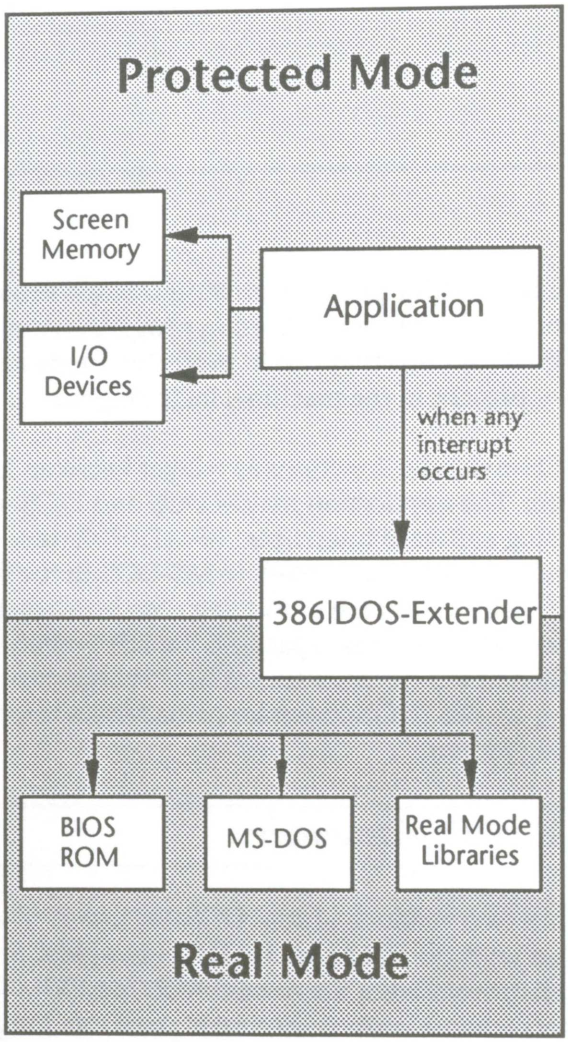


FIGURE 1-1
386|DOS-EXTENDER BLOCK DIAGRAM

Environments and Compatibility

386 | DOS-Extender runs on all 80386-, 80386SX-, and 80486-based PCs that are compatible with the three industry-standard architectures:

- ☛ IBM PC AT (also known as ISA, or Industry Standard Architecture)
- ☛ IBM PS/2 Micro Channel Architecture (MCA)
- ☛ the Extended Industry Standard Architecture (EISA)

In addition to running under MS-DOS, 386 | DOS-Extender is compatible with Microsoft Windows 3.0 in real and standard modes, Quarterdeck DESQview 386, and all EMS emulators that support the VCPI interface. Version 3.0 of 386 | DOS-Extender is not compatible with Windows 3.0 enhanced mode, or with the DOS boxes provided in OS/2 or UNIX.

DPMI (DOS Protected-Mode Interface) is an interface provided in Windows 3.0 enhanced mode. It will likely be supported in future versions of OS/2 and UNIX. Although version 3.0 of 386 | DOS-Extender does not support DPMI, Phar Lap believes DPMI is very important and is committed to adding DPMI support. Because DPMI will not allow all 386 | DOS-Extender functionality to be supported, this manual identifies features that will not be provided under DPMI. If you are interested in the greater interoperability offered by DPMI, you may want to restrict your use of 386 | DOS-Extender features to those that will be available under DPMI.

Memory Terminology

386 | DOS-Extender allocates memory from all possible memory sources for use by the application. The following memory terms are used throughout the manual.

The term "conventional memory" refers to memory below one MB that is obtained from MS-DOS. The term "extended memory" refers to memory from all other sources.

There are several possible sources for extended memory. "XMS memory" refers to memory allocated through the XMS (eXtended Memory

Specification) interface, the Microsoft standard for allocation of memory above one MB. The XMS interface is usually provided by installing a device driver called HIMEM.SYS.

"VCPI memory" refers to memory allocated through the VCPI (Virtual Control Program Interface). VCPI is provided by most 386 EMS emulators (EMS is the Lotus-Intel-Microsoft [LIM] Expanded Memory Specification). Three well-known EMS emulators that support VCPI are the Microsoft EMM386, the Quarterdeck QEMM-386, and the Qualitas 386MAX.

"Direct extended memory" refers to memory above one MB that is not used by any other program and is not available through XMS or VCPI. 386 | DOS-Extender allocates this memory directly, using standard conventions described in Section 5.2.2.

Related Products

In addition to the products that comprise the 386 | DOS-Extender Software Development Kit, Phar Lap offers several related products:

☛ 386|VMM	A virtual memory system for use with 386|DOS-Extender, this product allows large applications to run on PCs with smaller amounts of memory.
☛ 386|SRCBug	A source level debugger for use with MetaWare High C, Zortech C++, and other compilers that provide CodeView symbolic information.
☛ 286|DOS-Extender	A 16-bit protected mode runtime environment for 80286-, 80386-, and 80486-based PCs.
☛ 386|DOS-Extender for NEC	A version of 386|DOS-Extender compatible with the Japanese NEC 9800 series of PCs.
☛ 386|ASM/LinkLoc	An embedded systems development toolkit for 80*x*86-based systems.

Chapter 2

Using 386 | DOS-Extender

2.1 Building Protected-Mode Programs

Programs to be run under 386 | DOS-Extender must be built for 80386 protected mode. The source code must be compiled or assembled with the 80386 as the target, and the object files must be linked for 80386 protected mode.

High-level language source code files must be compiled with a compiler that generates 80386 object code. Assembly language source code files must be assembled with an assembler such as the Phar Lap assembler, 386 | ASM, to produce 80386 object code. For example, to assemble a file named HELLO.ASM for the 80386, creating an object file named HELLO.OBJ, type:

```
386asm hello
```

Once the object files have been built, they are linked by the Phar Lap linker, 386 | LINK, to produce a protected-mode executable file. For example, to link a file named HELLO.OBJ with a library named CLIB.LIB, creating a protected-mode executable file named HELLO.EXP, type:

```
386link hello -lib clib
```

The HELLO.EXP file created by this example can be executed in 80386 protected mode under 386 | DOS-Extender. 386 | DOS-Extender can also be used to run executable files in .REX format (an alternative protected-mode file format) that have been built with 386 | LINK. For a complete description of object and executable file formats, please see the *386 | LINK Reference Manual.*

2.2 Command Line Syntax

There are two versions of the command line syntax used to run protected-mode programs under 386 | DOS-Extender. One is for running the program with the development version of 386 | DOS-Extender, the version provided with the 386 | DOS-Extender Software Development Kit. The other is for the runtime version provided when a 386 | DOS-Extender Redistribution Kit is purchased.

The way the program is run from the command line is the only difference between the development and runtime versions of 386 | DOS-Extender. The sections below describe the command line syntax for each version.

2.2.1 Development Version

The development version of 386 | DOS-Extender is distributed with the 386 | DOS-Extender Software Development Kit as a file named RUN386.EXE.

The command line syntax for RUN386 is as follows:

1. the name of the 386|DOS-Extender program file (RUN386), followed by any command line switches to be processed by 386|DOS-Extender, overriding its default operation
2. the name of the protected-mode program to run, i.e., the name of the .EXP or .REX file to be executed
3. any command line parameters to be passed to the protected-mode program.

In brief, using variable names to represent the above, the command line looks like this:

```
run386 switches pgmname params
```

If no file name extension is given for the protected-mode program, the default extension .EXP is assumed.

For example, to run the protected-mode program PECHO.EXP, which echoes its command line parameters, type:

```
run386 pecho Hello from protected mode!
```

To run PECHO.EXP with a 386 | DOS-Extender command line switch that will force the program to be loaded in memory above 1 MB:

```
   run386 -maxreal ffffh pecho Hello from above 1 MB!
```

2.2.2 Runtime Version

The runtime version of 386 | DOS-Extender is supplied when a 386 | DOS-Extender Redistribution Kit is purchased to allow redistribution of 386 | DOS-Extender with a protected-mode application. It is distributed as a file named RUN386B.EXE.

This RUN386B.EXE file must be "bound" to the protected-mode application using the binder utility called BIND386, supplied when the 386 | DOS-Extender Redistribution Kit is purchased. Binding combines the RUN386B.EXE file with the protected-mode program file. This produces a single .EXE file, which can be run in the standard way (by typing the file name) from the MS-DOS command prompt. The binding process is described in detail in the *BIND386 Utility Guide*.

The following example binds a protected-mode application named PECHO.EXP with RUN386B.EXE to produce a file named PECHO.EXE:

```
   bind386 run386b pecho
```

The bound program PECHO.EXE is then run in the standard MS-DOS fashion, by typing the file name:

```
   pecho Hello from protected mode!
```

With bound application programs, the entire command line is passed to the protected-mode application for processing. If it is necessary to use 386 | DOS-Extender command line switches, they must be entered with an environment variable, or configured into the bound program with the CFIG386 utility (please see section 2.3).

Phar Lap recommends that you give end users of your application the capability to set some 386 | DOS-Extender switches. The runtime license gives you the right to ship CFIG386.EXE with your bound application. You may also reproduce some or all of the *CFIG386 Utility Guide* and sections 2.3

through 2.12 of this manual in your product documentation. The file SWITCHES.TXT on the 386 | DOS-Extender Redistribution Kit product disk contains descriptions of switches suitable for use by end users, and may be redistributed with your product.

The recommended method for allowing end users to set switches is to use an environment variable you select for your product. This is simpler for end users, because there is no need to instruct them in the use of CFIG386.

For example, if you select the environment variable MYENVAR for your product, you configure your program as follows:

```
cfig386 pecho %myenvar
```

To set 386 | DOS-Extender switches, the end user just enters the environment variable. For example, to load the program in memory above 1 MB, the user would enter the following:

```
set myenvar=-maxreal ffffh
pecho
```

You should not tell end users to set switches with the DOSX environment variable that 386 | DOS-Extender always reads, because then the switches would apply to any 386 | DOS-Extender-based program the user runs, rather than specifically to your application.

If you prefer not to use an environment variable, you can give end users the ability to set switches by including CFIG386.EXE and documentation on its use.

2.3 Command Line Switches

Command line switches are used to change the default operation of 386 | DOS-Extender. They begin with a minus sign (-) character, followed by the name of the switch. There is a long and a short form for each switch name. Any arguments to the switch must immediately follow the switch name, separated by a space. If conflicting switches are given on a command line, the last (rightmost) switch takes precedence.

Some command line switches take a number as an argument. By default, the number is considered to be decimal (base 10).

To specify hexadecimal (base 16) numbers, append the character "h" or "H" to the number. For example, the numbers in the examples below are equivalent as arguments to the switch -MAXREAL:

```
run386 -maxreal 200h hello

run386 -maxreal 512 hello
```

386 | DOS-Extender switches may be provided in four different ways:

☛ Specified when the program is linked. These include only the following switches:

-MINREAL	-NISTACK	-PRIVILEGED	-REALBREAK	-MINIBUF
-MAXREAL	-ISTKSIZE	-UNPRIVILEGED	-CALLBUFS	-MAXIBUF

☛ Specified in an environment variable named DOSX

☛ Configured into the 386|DOS-Extender program file (RUN386.EXE or RUN386B.EXE) using the CFIG386 utility

☛ Entered on the command line when the program is actually run.

Link–time switch settings are processed first; then switches in the DOSX environment variable; then configured-in switches. Command line switches are processed last. If conflicting switch settings are given, the last switch processed takes precedence. The following examples illustrate the four ways of providing 386 | DOS-Extender switches:

```
386link hello -maxr ffffh
run386 hello

set dosx=-maxr ffffh
run386 hello

cfig386 run386 -maxr ffffh
run386 hello

run386 -maxr ffffh hello
```

For distribution to end users, you have two additional ways to customize your application to look for 386 | DOS-Extender switches: in an arbitrary environment variable or a command file on disk. Sections 2.3.1 and 2.3.2 describe these methods.

After section 2.3.2, the remaining sections of this chapter describe the 386 | DOS-Extender command line switches, using the following categories:

- Memory Management Switches (section 2.4)
- Privilege Level Switches (section 2.5)
- Mixed–Mode Program Switches (section 2.6)
- GDT and LDT Size Switches (section 2.7)
- Coprocessor Switches (section 2.8)
- Stack Allocation Switches (section 2.9)
- 386/387 Paging Errata Workarounds (section 2.10)
- 386|VMM Switches (section 2.11)
- Rarely Used Switches (section 2.12)

2.3.1 Environmental Variable Usage

Command line switches can be placed in an arbitrary environment variable, in addition to the hardwired environment variable DOSX. You specify your environment variable name as *%varname*. If the specified environmental variable does not exist, it is simply ignored. This capability allows an application program to be configured to scan an environment variable unique to that application. For example:

```
bind386 run386b hello
cfig386 hello %myswitches
set myswitches=-maxr ffffh
hello
```

2.3.2 Command File Usage

Conditional command files can be specified using the syntax @@*filename*. If no file extension is given, a default extension of .CMD is assumed. If a file path is not given, the file must be in the current default directory. If the specified command file is not present, 386 | DOS-Extender continues without signalling an error. This capability allows an application program to be configured to process a command file unique to that application.

For example, if the file MYDOSX.CFG contains -maxr ffffh, you can configure 386 | DOS-Extender to read the file with these commands:

```
bind386 run386b hello
cfig386 hello @@mydosx.cfg
hello
```

2.4 Memory Management Switches

386 | DOS-Extender use of all memory resources can be controlled through its command line switches. The memory management switches fall into three broad categories of control:

- conventional memory (section 2.4.1)
- extended memory (section 2.4.2)
- applications' linear memory usage (section 2.4.3).

For more information on memory management, please see Chapter 5.

2.4.1 Conventional Memory Switches

The -MINREAL and -MAXREAL switches control how much conventional memory (memory below 640 KB) is left free by 386 | DOS-Extender when the application program is loaded.

The -MINIBUF and -MAXIBUF switches control how much memory is allocated to the data buffer used for DOS and BIOS function calls.

Free Memory Switches (-MINREAL and -MAXREAL)

By default, 386 | DOS-Extender allocates all the available conventional memory for use by the application program. Please see section 5.2.1 for a description of how this memory is used.

You may want to leave conventional memory free if conventional memory will be needed either by your program or by another program in the system. For example, if your program has a feature to run an arbitrary DOS program, you will need to leave enough conventional memory free to satisfy the requirements of the DOS command interpreter and the program to be run. An alternative to using these switches to leave memory free at load time is to use the 2525h or 2536h system calls (please see Appendix B) at run time to adjust conventional memory usage.

-MINREAL specifies the minimum amount of conventional memory to leave free. 386 | DOS-Extender refuses to run the program if it cannot leave at least this amount of memory free.

-MAXREAL specifies the maximum amount of conventional memory to leave free. 386 | DOS-Extender guarantees that at least the MINREAL memory is left free, and that as much as possible (up to the MAXREAL memory) is left free.

Both -MINREAL and -MAXREAL take a number up to 65535 (FFFFh) as an argument, specifying memory size in units of 16-byte paragraphs. This is the standard unit of memory allocation under MS-DOS. By default, 386 | DOS-Extender sets both -MINREAL and -MAXREAL to zero. These switches may also be specified at program link time.

Syntax:

```
-MINREAL nparagraphs
-MAXREAL nparagraphs
```

Short Form:

```
-MINR nparagraphs
-MAXR nparagraphs
```

Examples:

```
run386 -minreal 100h hello
run386 -minr 128 -maxr 512 hello
```

DOS Data Buffer Switches (-MINIBUF and -MAXIBUF)

The data buffer used for DOS and BIOS function calls is important for file I/O. If your program reads or writes large amounts of data at a time, use -MINIBUF and -MAXIBUF to allocate a large buffer for efficiency.

Both switches take a number between 1 and 64 (inclusive) as an argument, specifying the buffer size in units of 1 KB. By default, 386 | DOS-Extender sets MINIBUF to 1 KB, and MAXIBUF to 16 KBs. If 386 | DOS-Extender cannot allocate at least MINIBUF KBs for the data buffer, it refuses to run the program.

If possible, MAXIBUF KBs are allocated. If there is not enough memory available to satisfy both the MAXREAL and MAXIBUF parameters, MAXIBUF takes precedence. These switches may also be specified at program link time.

Syntax:

```
-MINIBUF nkilobytes
-MAXIBUF nkilobytes
```

Short Form:

```
-MINI nkilobytes
-MAXI nkilobytes
```

Examples:

```
run386 -maxibuf 2 hello
run386 -mini 64 -maxi 64 filecopy
```

2.4.2 Extended Memory Control

Three sets of switches control extended memory allocation and usage, as explained in the subsections that follow. The first set described limits physical memory usage:

-MAXE[XTMEM]	specifies maximum bytes of direct extended memory to allocate
-MAXV[CPIMEM]	specifies maximum bytes of VCPI memory to allocate
-MAXX[MSMEM]	specifies maximum bytes of XMS memory to allocate
-MAXBL[KXMS]	specifies maximum bytes in any individual XMS block

The second set described controls compatibility with programs that don't mark extended memory usage:

-EXTL[OW]	establishes lowest address in direct extended memory that the application can use
-EXTH[IGH]	establishes highest address in direct extended memory that the application can use

The third set described relates to COMPAQ built-in memory:

-NOSPCL[MEM]	tells 386 | DOS-Extender not to look for special memory
-NOB[IM]	Synonym for NOSPCLMEM

Under DPMI, all extended memory is allocated through the DPMI interface, and these switches are ignored.

Limiting Physical Memory Usage

By default, 386 | DOS-Extender allocates as much memory as the application program requests, up to all available memory from all memory sources. In a multitasking environment such as Windows or DESQview, it may be desirable to limit an application's physical memory consumption, to leave memory free for other programs.

The -MAXBLKXMS switch defines the maximum size, in bytes, of an individual XMS block (default = 4 GB). XMS memory, unlike direct extended memory and VCPI memory, cannot be released until after the program terminates. This switch can be used to cause 386 | DOS-Extender to allocate XMS memory in smaller chunks, leaving some XMS memory free if the application doesn't need it all.

The -MAXXMSMEM switch defines the maximum number of bytes of XMS memory to allocate (default = 4 GB).

The -MAXEXTMEM switch defines the maximum number of bytes of direct extended memory to allocate (default = 4 GB).

The -MAXVCPIMEM switch defines the maximum number of bytes of memory to be allocated from an EMS emulator that supports the VCPI interface (default = 4 GB).

Syntax:

```
-MAXBLKXMS nbytes
-MAXXMSMEM nbytes
-MAXEXTMEM nbytes
-MAXVCPIMEM nbytes
```

Short Form:

```
-MAXBL nbytes
-MAXX nbytes
-MAXE nbytes
-MAXV nbytes
```

Examples:

```
run386 -maxbl 80000h hello
run386 -maxx 100000h hello
run386 -maxe 600000h hello
run386 -maxv 600000h hello
```

Compatibility with Programs Not Marking Memory Usage

The -EXTLOW and -EXTHIGH switches limit the amount of direct extended memory that 386 | DOS-Extender allows the application program to use. By default, all direct extended memory not allocated to other programs is available for use by the application. Other programs that may have allocated direct extended memory include RAM disk programs, disk cache programs, and EMS emulators.

Normally, it is not necessary to use the -EXTLOW or -EXTHIGH switches. However, if your system has a program installed that uses direct extended memory, the program should use one of the following two methods (please see sections 5.2.2 and 8.9.1) for allocating memory above one megabyte:

- the VDISK or RAMDRIVE standards for allocating memory from 1 MB up
- the INT 15h function 88h BIOS call for allocating extended memory from the top of memory down.

If installed programs use neither of these methods, -EXTLOW or -EXTHIGH (or both) may be necessary to prevent 386 | DOS-Extender from allocating extended memory used by the installed program. Please see Chapter 5 for more details on memory allocation, and Chapter 8 regarding compatibility issues.

Both -EXTLOW and -EXTHIGH take a number as an argument, specifying a physical memory address in extended memory. By default, 386 | DOS-Extender sets EXTLOW to 100000h (1 MB) and EXTHIGH to FFFFFFFFh (4 GB).

386 | DOS-Extender uses only extended memory above the address specified with the -EXTLOW switch, or memory used by other programs, whichever is higher. Similarly, it uses only extended memory below the address specified with the -EXTHIGH switch, or memory used by other programs, whichever is lower.

Syntax:

```
-EXTLOW address
-EXTHIGH address
```

Short Form:

```
-EXTL address
-EXTH address
```

Examples:

```
run386 -extlow 200000h hello
run386 -extl 180000h -exth 400000h hello
```

Adjusting for Special Memory

The -NOSPCLMEM switch disables the automatic use of special memory, such as the built-in memory mapped above 14 MB on COMPAQ 386 machines. By default, 386 | DOS-Extender attempts to use such memory if it is not allocated to another program. This switch (synonym: -NOBIM) instructs 386 | DOS-Extender not to check for such memory. Normally, it is not necessary to use this switch.

Syntax:

```
-NOSPCLMEM
-NOBIM
```

Short Form:

```
-NOSPCL
-NOB
```

Example:

```
run386 -nobim hello
```

2.4.3 Limiting Linear Memory Usage by Application Programs

The -MAXP[GMMEM] switch defines the maximum amount of linear memory the application program can allocate, with a default of 4 GB.

If not using 386 | VMM, the -MAXP switch limits the total amount of physical memory consumed by the application program, since all linear memory pages must be in physical memory without 386 | VMM.

If using 386 | VMM, the -MAXP switch limits the total amount of physical memory plus disk space for the swap file consumed by the application program.

Syntax:

```
-MAXPGMMEM nbytes
```

Short Form:

```
-MAXP nbytes
```

Example:

```
run386 -maxp 500000h hello
```

2.5 Privilege Level

The -PRIV and -UNPRIV switches establish the privilege level for the application's execution: privileged (level 0) or unprivileged (levels 1, 2, or 3). The default is privileged. This switch can be set at link time with 386 | LINK.

Privilege level is also known as ring level: 0 is the most privileged level of processor information; 3 is the least privileged. Please see sections 3.1, 3.2, and 4.1 for more information on privilege levels.

Privileged (level 0) operation is provided for backward compatibility with earlier versions of 386 | DOS-Extender. Unprivileged operation is strongly recommended. Privilege level 3 is always used for unprivileged operation in a non-DPMI environment.

Under DPMI, privileged operation cannot be supported. If you use -PRIV, 386 | DOS-Extender will refuse to run your program in a DPMI environment. In addition, under DPMI the actual privilege level is determined by the DPMI host and may be either 1, 2, or 3. For DPMI compatibility, your application should not assume a specific privilege level.

Syntax:

```
-PRIVILEGED
-UNPRIVILEGED
```

Short Form:

```
-PRIV
-UNPRIV
```

Example:

```
run386 -priv myprog
run386 -unpriv hello
```

2.6 Mixed-Mode Program Switches

The -REALBREAK and -CALLBUFS switches control the program loading of programs that contain both real-mode and protected-mode code. This topic is discussed in detail in Chapter 7.

-REALBREAK controls how much of the program must be loaded into conventional memory for access and/or execution in real mode. It takes an argument specifying the number of bytes at the beginning of the program, which must be loaded in conventional memory. This switch may also be specified at link time. It is usually more convenient to specify this switch at link time, when the argument can be the name of a public symbol appearing at the end of the real-mode code and data. If this switch is used at run time, the argument must be an absolute number calculated from the information in the link map.

-CALLBUFS controls the size of the intermode call buffer, which is allocated in conventional memory. The buffer is used by the application program as a data buffer on intermode procedure calls. The buffer address is obtained at run time with the 386 | DOS-Extender system call 250Fh (please see Appendix B).

The CALLBUFS argument is the size of the buffer in KBs; it must be less than or equal to 64. The default buffer size is zero. This switch may also be specified at link time.

Note: Under a DPMI environment, -REALBREAK requires the optional Conventional Memory capability (please see Appendix F), which is not available under Windows 3.0. If this capability is not present, 386 | DOS-Extender prints an error message and refuses to run the program.

Syntax:

```
-REALBREAK nbytes
-CALLBUFS nkilobytes
```

Short Form:

```
-REALB nbytes
-CALLB nkilobytes
```

Examples:

If the public symbol END_REAL marks the end of the program's real mode code and data, use the -REALBREAK switch at link time:

```
386link switch -realb END_REAL -callbufs 2
```

Assuming the program map file shows END_REAL at offset 2010h, an alternative method is specifying the -REALBREAK value at run time:

```
run386 -realbreak 2010h -callb 2 switch
```

2.7 GDT and LDT Size Switches

The GDT and LDT each have an initial size of 4 KBs (512 segment descriptors), which can be increased by using the -GDTENT and -LDTENT switches. Each takes a number as the argument, representing the desired number of descriptors to allocate.

The Allocate Segment system call (INT 21h, function 48h) dynamically increases the LDT size if there are no free segments available in the LDT.

Some application programs depend on allocating specific segment descriptors in either the GDT or the LDT, that is, assigning segment selectors at link time. Such a program should always use segment selectors at or above 800h for the GDT and 804h for the LDT. The first 256 segments in both the GDT and the LDT should not be used; they are reserved for use by future versions of 386 | DOS-Extender.

Syntax:

```
-GDTENT nentries
-LDTENT nentries
```

Short Form:

```
-GDTE nentries
-LDTE nentries
```

Examples:

```
run386 -gdte 600 -ldte 1200 hello
```

2.8 Coprocessor Switches (-WEITEK and -CYRIX)

The -WEITEK switch selects how detection of the Weitek 1167, 3167, or 4167 floating point coprocessor is performed. If 386 | DOS-Extender detects the presence of a Weitek coprocessor, segment selector 003Ch is initialized to map the memory space used by the Weitek coprocessor and segment register FS is initialized to contain selector 003Ch. A program can, therefore, test for the presence of a Weitek coprocessor at run time by examining the contents of the FS register. Section 4.5 contains more information on programming the Weitek coprocessor.

The -WEITEK switch has three settings. The -WEITEK AUTO setting instructs 386 | DOS-Extender to use the Weitek-approved BIOS presence detection call. This may not work correctly on all machines, if the appropriate BIOS is not installed. The setting -WEITEK ON instructs 386 | DOS-Extender to assume the Weitek coprocessor is present, and setting -WEITEK OFF assumes the Weitek coprocessor is not present. The default setting is -WEITEK AUTO.

Syntax:

```
-WEITEK AUTO
-WEITEK ON
-WEITEK OFF
```

Short Form:

```
-WEITEK AUTO
-WEITEK ON
-WEITEK OFF
```

Example:

```
run386 -weitek on float.exp
```

The -CYRIX switch, like -WEITEK, controls detection of the Cyrix EMC87 coprocessor. (The EMC87 executes normal 387 opcodes and can also execute faster via a memory-mapped interface.) The default is -CYRIX AUTO.

Cyrix automatic detection is done by attempting to toggle a bit in the control word that cannot be toggled on a 387 chip. This is an extremely reliable detection method, and it should not normally be necessary to use -CYRIX ON or -CYRIX OFF. At run time, a program can determine whether the Cyrix is present with the Get Configuration Information system call (function 2526h). If the Cyrix coprocessor is present, segment 004Ch is initialized to map the memory space used for the high-performance memory-mapped interface.

Syntax:

```
-CYRIX AUTO
-CYRIX ON
-CYRIX OFF
```

Short Form:

```
-CYRIX AUTO
-CYRIX ON
-CYRIX OFF
```

Example:

```
run386 -cyrix on float.exp
```

2.9 Stack Allocation Switches

The -NISTACK and -ISTKSIZE switches control how much memory is allocated to the buffers used to provide stack space when switching the 80386 from protected mode to real mode. For a complete description of the issues involved in allocating stack buffers, please see section 7.4. For most application programs, the default settings of these parameters are sufficient.

Both switches take a number as an argument. The -NISTACK switch specifies the number of stack buffers to allocate and must be six or greater. The -ISTKSIZE switch specifies the size of each stack buffer in KBs and must be between 1 and 64, inclusive. By default, 386 | DOS-Extender allocates six stack buffers of 1 KB each. These switches may also be specified at program link time.

Syntax:

```
-NISTACK nbuffers
-ISTKSIZE nkilobytes
```

Short Form:

```
-NI nbuffers
-IS nkilobytes
```

Example:

```
run386 -ni 8 -istk 2 switch.exp
```

2.10 386/387 Paging Errata Workarounds (-NOPAGE and -ERRATA17)

80386 chip steps B1 and earlier have two chip errata (known as "erratum 17" and "erratum 21") that occur only with protected mode programs that use the 80387 floating point coprocessor. Both errata cause the 386 to stop processing instructions when an 80387 instruction is executed under certain conditions (i.e., the program appears to hang). Neither erratum exists in any 386SX chip, or in 386 chip steps D0 (released in the second quarter of 1988) and later.

Erratum 21 depends on a number of conditions, including hardware timing. The problem may go away when the program is rebuilt (resulting in different alignment of 387 instructions), and even the same .EXP file often hangs in different program locations in successive program runs. A symptom of erratum 21 is that the machine cannot be rebooted with CTRL-ALT-DEL; the power must be switched off.

Erratum 17 depends on instruction alignment, and only occurs when the program is running with 386 | VMM. Rebuilding the application often makes the problem disappear temporarily; successive runs with the same .EXP file often hang in different places because of different paging behavior. When a programs hangs because erratum 17 occurs, it is possible to reboot with CTRL-ALT-DEL.

One of the conditions required for both erratum 17 and erratum 21 is that the paging hardware of the 386 be enabled. The -NOPAGE switch disables paging to work around the errata. However, there are a number of drawbacks to using -NOPAGE; please see section 2.10.1.

The -ERRATA17 switch installs a software workaround for erratum 17, but leaves paging enabled (there is no viable software workaround for erratum 21 with paging enabled). The -ERRATA17 switch is only needed if 386 | VMM is used, and is only effective if a hardware workaround for erratum 21 is in place (preventing erratum 17 does no good if erratum 21 can still occur). Please see section 2.10.2 for details.

2.10.1 Paging Disable Switch

The -NOPAGE switch forces 386 | DOS-Extender to run with the 386 hardware paging capability disabled. This is a workaround for both erratum 17 and erratum 21, by removing one of the conditions needed for either erratum to occur. However, there are a number of serious drawbacks to using -NOPAGE:

- If -NOPAGE is used, 386|DOS-Extender refuses to run the program in a VCPI (EMS emulator) or DPMI (such as Windows 3.0 enhanced mode) environment.
- 386|VMM cannot be used with -NOPAGE.
- The -REALBREAK switch and the 386|LINK -OFFSET switch cannot be used.
- Conventional memory and COMPAQ built-in memory cannot be used by the application (only direct extended memory and XMS memory are available to the application).
- Memory management services are severely curtailed. 386|DOS-Extender system calls 250Ah, 250Fh, 251Ah, 251Bh, 251Ch, 251Dh, 251Eh, 251Fh, 2521h, 2525h, 252Bh, 252Ch, and 2536h cannot be used. The Resize Segment DOS

call (function 4Ah) works only in a limited way; memory freed by reducing segment size cannot always be reallocated.

☛ Many debuggers do not work well with -NOPAGE, because of the restrictions in memory management capabilities. It may be necessary to debug without -NOPAGE on a machine that has a hardware workaround or a chip step without the errata.

If possible, use a hardware workaround for these errata instead of the -NOPAGE switch. Use chip step D0 or later of the 386 chip, or get a hardware piggyback board with a workaround for erratum 21 from Intel or other hardware vendors. Some PC motherboards have a hardware workaround for erratum 21 designed in; for example, all COMPAQ 386/20 PCs have a hardware workaround.

If you use a machine with a B1 386 chip and a hardware workaround for erratum 21, and your program uses 386 | VMM, use the -ERRATA17 switch (please see section 2.10.2) to avoid erratum 17.

Syntax:

```
-NOPAGE
```

Short Form:

```
-NOP
```

Example:

```
run386 -nopage numcrunch
```

2.10.2 80386 Erratum 17 Workaround

The -ERRATA17 switch installs a software workaround for erratum 17 of the 386 chip. Use this switch when running under 386 | VMM on a machine with a B1 step 386 chip and a hardware workaround for erratum 21. There is no point in using -ERRATA17 on a machine with a B1 386 chip and no hardware fix for erratum 21.

The only drawback with using the -ERRATA17 switch is a small number of additional instructions executed on every timer tick. You may want to consider always configuring -ERRATA17 into bound executables you send to your customers.

Syntax:

```
-ERRATA17
```

Short Form:

```
-ERR
```

Example:

```
run386 -nopage numcrunch
```

2.11 386 | VMM Switches

The following switches apply only to operation with 386 | VMM, and are explained in the *386 | VMM Reference Manual.* These switches are all ignored (have no effect) if 386 | VMM is not present.

-CODESIZE *nbytes*
-DEMANDLOAD
-FLUSHSWAP
-LFU
-LOCKSTACK *nbytes*
-MAXSWFSIZE *nbytes*
-MINSWFSIZE *nbytes*
-NOPGEXP
-NOSWFGROW1ST
-NOVM
-NUR
-PAGELOG *filename*
-SWAPCHK ON | OFF | FORCE | MAX
-SWAPDEFDISK
-SWAPDIR *dirname*
-SWAPNAME *filename*
-SWFGROW1ST
-VMFILE *filename*
-VSCAN *nmilliseconds*
-VSLEN *nbytes*

2.12 Rarely Used Switches

The switches in this section are generally not used, since they deal with relatively rare compatibility issues. They are discussed in the order of the following categories:

- Hardware Architecture Switches
- Address Line 20 Switch
- VDISK Compatibility Switch
- Save 32-bit Registers Switch
- Disable Multiply Check Switch
- Don't Run Under VCPI Switch
- Debug Printout Switches
- System Call Pointer Conversion Switch
- Interrupt Control Switches
- BIOS Block Move Switch
- Open .EXP File in Write Deny Mode Switch

2.12.1 Hardware Architecture Switches

386 | DOS-Extender normally correctly identifies the architecture of the PC on which the program is run. For compatibility, switches are provided to override the automatic detection capability of 386 | DOS-Extender. These switches include the -XT, -AT, -EISA, and -MCA switches, identifying the hardware in use.

XT Detection Switch

The -XT switch informs 386 | DOS-Extender that it is executing on an IBM-compatible PC or PC/XT with a 386 board, such as the Intel Inboard-PC, installed. 386 | DOS-Extender normally detects such configurations automatically, but it may not be able to detect systems that do not have the IBM-standard system ID byte in the BIOS. If 386 | DOS-Extender does not correctly detect a PC environment, this switch can be used to allow the program to execute successfully.

Syntax:

```
-XT
```

Short Form:

```
-XT
```

Example:

```
run386 -xt hello
```

AT Detection Switch

The -AT switch, taking no arguments, overrides the 386 | DOS-Extender detection algorithm for the computer architecture on which it is running. It forces selection of the IBM PC AT (ISA) bus architecture.

Syntax:

```
-AT
```

Short Form:

```
-AT
```

Example:

```
run386 -at hello
```

EISA Detection Switch

The -EISA switch, taking no arguments, overrides the 386 | DOS-Extender detection algorithm for the computer architecture on which it is running. It forces selection of the EISA architecture.

Syntax:

```
-EISA
```

Short Form:

```
-EISA
```

Example:

```
run386 -eisa hello
```

MCA Detection Switch

The -MCA switch, taking no arguments, overrides the 386 | DOS-Extender detection algorithm for the computer architecture on which it is running. It forces selection of the IBM PS/2 (Micro Channel) bus architecture.

Syntax:

```
-MCA
```

Short Form:

```
-MCA
```

Example:

```
run386 -mca hello
```

2.12.2 Address Line 20 Switch

The -A20 switch controls how address line 20 is enabled or disabled.

80386 systems that conform to the IBM PC AT standard have hardware either to allow full 32-bit addressing ("enable A20") or to truncate addresses to 20 bits ("disable A20"). When executing in real mode, A20 is normally disabled for compatibility with programs that take advantage of the address space wrap-around occurring at 1 MB on 8088-8086 systems. Very few programs rely on this behavior; the most common examples are copy protection programs.

By default, 386 | DOS-Extender enables A20 before starting the application, and restores the original A20 setting when the program terminates. The -A20 switch can be used to force 386 | DOS-Extender to disable A20 each time the 80386 switches to real mode, and to re-enable A20 each time the 80386 switches to protected mode. This can be important if, for example, a software driver, which can gain control at any time via a hardware interrupt, and which relies on 1 MB addressing wrap-around, is installed on your machine.

There is a penalty associated with the -A20 switch. Depending on the hardware in your system, it can take several milliseconds to enable or disable A20. Thus, using the -A20 switch slows down the switch to 80386 real mode, then back to protected mode, that occurs whenever there is a hardware interrupt or a DOS or BIOS function call.

Syntax:

```
-A20
```

Short Form:

```
-A20
```

Example:

```
run386 -A20 hello
```

2.12.3 VDISK Compatibility Switch

The -VDISK switch is a workaround for compatibility problems with other programs that do not correctly follow the VDISK standard for allocating extended memory. The problem is described in section 8.1.

If 386 | DOS-Extender refuses to run an application program because of inconsistent VDISK allocation signatures, this switch can be used to force 386 | DOS-Extender to run the program. The larger of the two allocation marks present will be used. Before using this switch, you should check the allocation sizes printed out with the error message when 386 | DOS-Extender refuses to run the program. If the larger of the two numbers printed out does not seem reasonable, it will be necessary to calculate how much extended memory is in use by other programs and to use the -EXTLOW switch to inform 386 | DOS-Extender of the correct value.

Syntax:

```
-VDISK
```

Short Form:

```
-VDISK
```

Example:

```
run386 -vdisk hello
```

2.12.4 Save 32-bit Registers Switch

The -SAVEREGS switch forces 386 | DOS-Extender to preserve the high 16 bits of general registers across switches to real mode initiated by software interrupts. It uses no arguments.

Using the -SAVEREGS switch protects an application program from real-mode code that uses but does not restore the high 16 bits of the general registers. This register saving operation is always performed by default for INT 21h and INT 15h.

This switch has no effect on mode switches initiated by system calls 250Eh, 2510h, or 2511h.

Syntax:

```
-SAVEREGS
```

Short Form:

```
-SAVER
```

Example:

```
run386 -saver hello
```

2.12.5 Disable Multiply Check Switch

The -NOMUL switch, taking no arguments, prevents 386 | DOS-Extender from checking at startup time for 386 chips that exhibit the 32-bit multiply problem. This problem can cause incorrect results from a 32-bit MUL instruction.

Normally, if 386 | DOS-Extender detects a chip with the problem, it refuses to run and prints an error message. This switch forces 386 | DOS-Extender to run anyway. 386 | DOS-Extender never uses the instruction that gives erroneous results.

Syntax:

```
-NOMUL
```

Short Form:

```
-NOMUL
```

Example:

```
run386 -nomul hello
```

2.12.6 Don't Run Under VCPI Switch

The -NOVCPI switch, taking no arguments, causes 386 | DOS-Extender to refuse to run if an EMS emulator with the VCPI interface is installed.

Syntax:

```
-NOVCPI
```

Short Form:

```
-NOVCPI
```

Example:

```
run386 -novcpi hello
```

2.12.7 Debug Printout Switches

The -DEBUG switch turns on debug printout in 386 | DOS-Extender.

The I/O redirection switches, -COM1, -COM2, and -BAUD, redirect output from the -DEBUG switch. They display to a terminal connected to communications port 1 or 2 on the back of the PC. This can be useful when using the -DEBUG switch with application programs that use the system display extensively; debug output will not interfere with the application's display setup.

Enabling Debug Printout

The -DEBUG switch causes 386 | DOS-Extender to print debug information on the display. It takes a debug level as an argument; the higher the level, the more verbose the output. Except where noted, each level includes all information printed at lower debug levels. Levels 0-3 only print information during initialization; levels 4-6 also print information while the application program is executing.

Level	Information
0	none
1	environment (processor, machine architecture, interfaces present and their version numbers, etc.) information, amount of memory available, application program size
2	more detailed memory size breakdown, information on conventional memory consumed by 386 | DOS-Extender
3	prints information about the internal initialization stages of 386 | DOS-Extender; this is a good level to use when calling Phar Lap about a problem that occurs before your program starts executing
4	prints register values when INT 21h system calls are made in protected mode
5	message printed each time a physical page is allocated or freed, and each time a page fault occurs under 386 | VMM, but INT 21h system calls are not printed
6	both INT 21h system calls and page faults/allocations are printed

Syntax:

```
    -DEBUG level
```

Short Form:

```
    -DEBUG level
```

Example:

```
    run386 -debug 1 hello
```

I/O Redirection Switches

The -COM1 switch redirects output from the -DEBUG switch to communications port 1; -COM2 to communications port 2. If either switch is used without the -BAUD switch, the port's operations mode is not modified.

You can use the DOS MODE command to modify the parameters of a communications port.

The -BAUD switch can be used with -COM1 or -COM2 to set the port operations mode. -BAUD takes a single argument: the baud rate for the port. Permissible values are 110, 150, 300, 600, 1200, 2400, 4800, or 9600. When -BAUD is used, the other port parameters are automatically set at 8 data bits, one stop bit, and no parity. Use of the -BAUD switch thus overrides any previous MODE command entered from MS-DOS.

Syntax:

```
-COM1
-COM2
-BAUD speed
```

Short Form:

```
-COM1
-COM2
-BAUD speed
```

Examples:

```
run386 -com1 -debug 1 hello
run386 -com2 -baud 9600 -debug 3 hello
```

2.12.8 System Call Pointer Conversion Switch

The -DATATHRESHOLD switch sets the data buffer size below which 386 | DOS-Extender will always buffer data on a DOS or BIOS system call. The alternative is attempting to convert the protected-mode pointer to the buffer to a real-mode pointer (please see Chapter 7).

-DATATHRESHOLD 0 forces 386 | DOS-Extender to always attempt pointer conversion before buffering the data. -DATATHRESHOLD 10000h (or larger) forces 386 | DOS-Extender to always buffer the data and never attempt pointer conversion.

The default value is set to 10000h, which forces data to always be buffered rather than attempting a pointer conversion. The reason is that a buffer copy is very quick, and a pointer conversion is not necessarily cheaper; it requires scanning page tables to see if the buffer resides in contiguous conventional memory.

Syntax:

```
-DATATHRESHOLD nbytes
```

Short Form:

```
-DATAT nbytes
```

Example:

```
run386 -datat 4000h hello
run386 -datat 0 hello
```

2.12.9 Interrupt Control Switches

The 386 | DOS-Extender switches that control interrupt vectors and relocation are -SAVEINTS, -PRIVEC,-HWIVEC, and -INTMAP.

Preserving Real-Mode Interrupt Vectors

This switch saves all 256 real-mode interrupt vectors before loading the application, and restores them after the application terminates. It can be useful when debugging an application, so on an abnormal program termination 386 | DOS-Extender will restore any interrupts taken over by your application. The cost of using this switch is an extra 1 KB of conventional memory that is consumed for buffering the real-mode interrupt vectors.

Syntax:

```
-SAVEINTS
```

Short Form:

```
-SAVEI
```

Example:

```
run386 -savei hello
```

Relocating the BIOS Print Screen Interrupt

The IBM BIOS uses INT 5 as the print screen function call. In protected mode, INT 5 is treated as a BOUND exception by 386 | DOS-Extender, because some compilers (such as MetaWare Professional Pascal) issue INT 5 to signal a BOUND exception.

By default, 386 | DOS-Extender makes the BIOS print screen function available in protected mode by issuing an INT 80h. The -PRIVEC switch selects a different interrupt vector, in the range 30h - FFh. The switch -PRIVEC 5 can be used to tell 386 | DOS-Extender not to make the BIOS print screen function available in protected mode. Please see also section 6.8.3.

Syntax:

```
-PRIVEC vector
```

Short Form:

```
-PRI vector
```

Example:

```
run386 -pri FFh hello
run386 -pri 5 hello
```

Relocating Hardware Interrupts IRQ0 - IRQ7

386 | DOS-Extender needs to know which interrupt vectors are used for hardware interrupts. Normally, if the default MS-DOS interrupts are not used, 386 | DOS-Extender can automatically determine this condition. However, if there is a compatibility problem, due to another program relocating hardware interrupts IRQ0-7 from their default DOS interrupt vectors of 08h-0Fh, the -INTMAP switch can be used to tell 386 | DOS-Extender where hardware interrupts IRQ0-7 are mapped.

For backward compatibility with earlier versions, 386 | DOS-Extender also supports its own remapping of the IRQ0-7 interrupts. The -HWIVEC switch specifies the starting interrupt vector to which the block of eight interrupts should be relocated. The use of this switch is unnecessary, and Phar Lap strongly discourages it. See Appendix G and Appendix L for more information on the use of -HWIVEC.

The -HWIVEC and -INTMAP switches are ignored when running under DPMI.

Syntax:

```
-INTMAP vector
-HWIVEC vector
```

Short Form:

```
-INTM vector
-HWI vector
```

Example:

```
run386 -intm 50h hello
run386 -hwivec 50h hello
```

2.12.10 BIOS Block Move Switch

The -NOBMCHK switch, taking no arguments, prevents 386 | DOS-Extender from checking whether BIOS block move calls (INT 15h function 87h) are attempting to move data to locations 386 | DOS-Extender has allocated in direct extended memory.

The block move call is used by programs such as disk caches and RAM disks to copy data to and from extended memory. 386 | DOS-Extender checks for a valid destination because Phar Lap has encountered buggy products that attempt to use memory they have not allocated, and the results can be rather confusing program behavior.

There is normally no reason to use this switch to defeat the check, because it is very cheap and does not affect system performance.

Syntax:

```
-NOBMCHK
```

Short Form:

```
-NOBMCHK
```

Example:

```
run386 -nobmchk hello
```

2.12.11 Open .EXP File in Write Deny Mode Switch

The -OPENDENY switch forces 386 | DOS-Extender to open the application .EXP file with MS-DOS sharing mode set to write denied (rather than the default of compatibility mode). This switch is provided only for backward compatibility with earlier versions of 386 | DOS-Extender that used write deny sharing mode, and should not normally be used. Please see Appendix L for details.

Syntax:

```
-OPENDENY
```

Short Form:

```
-OPEND
```

Example:

```
run386 -opend hello
```

Chapter 3

Program Environment

3.1 Program Organization

In 32-bit protected mode, segments can be up to 4 GB long, and an offset within a segment is 32 bits wide. Segments are controlled by two system tables maintained by 386 | DOS-Extender: a global descriptor table (GDT) and a local descriptor table (LDT).

Segment registers are loaded with values called segment selectors. A segment selector is a 16-bit value that contains three fields:

☛ A 1-bit field that selects either the GDT or the LDT.

☛ A 13-bit field used to index an entry in the selected descriptor table (GDT or LDT). Each descriptor entry in either table is eight bytes long. It gives the linear base address for the segment, the size of the segment, and status information specifying the segment type, privilege level, etc. Please see the *Intel 80386 Programmer's Reference Manual* for a complete description of descriptor table entries and segment selectors.

☛ A 2-bit field used to control access to privileged segments.

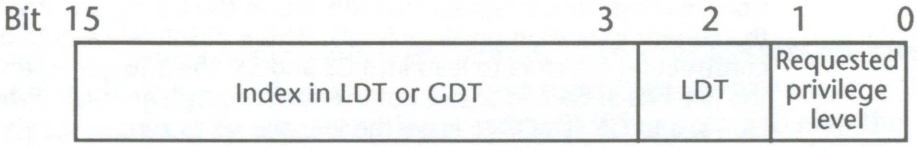


FIGURE 3-1
SEGMENT SELECTOR CONTENTS

386 | DOS-Extender uses the GDT to map its own segments and the LDT to map segments owned by the application program. The privilege level at which the program executes is carried in the two least-significant bits (LSBs) of the CS register (with zero the most privileged level). By default,

386 | DOS-Extender runs programs at level 0. Phar Lap recommends you use the -UNPRIV switch to run your programs at level 3. Making your program compatible with unprivileged operation will allow you to run in DPMI environments such as Windows 3.0 enhanced mode with a later 386 | DOS-Extender release that supports DPMI. Please see Appendix H for more information on level 0 operation.

Programs that run as unprivileged (-UNPRIV) should not assume that the actual privilege level will always be level 3. Under DPMI, the DPMI host selects the privilege level: 1, 2, or 3. For example, Windows 3.0 runs programs at level 1. So, segment selector values will differ accordingly. For example, the selector for the application's data segment at level 0 is 0014h; at level 1, 0015h; and at level 3, 0017h.

In this manual, hardwired segment selectors are always shown with their level 0 values for clarity (e. g., the program data segment always appears as 0014h, not as 0017h). Remember that the selector values seen by the application program will vary depending on privilege levels. Keep the following points in mind:

- Application segment selectors constructed by 386|DOS-Extender always have the Requested Privilege Level (RPL) bits set to the application's privilege level. This includes the initial segment register values at program startup, and any segment selectors returned by 386|DOS-Extender system calls.
- When checking a segment selector value, it's usually easiest to mask off the RPL bits before checking the value.
- When constructing a segment selector value, always OR in the RPL bits obtained from reading the CS register (the RPL bits in the CS register are always equal to the current execution privilege level). This is absolutely required when constructing selectors to load into CS and SS; the 386 generates an exception if the RPL bits in CS and SS are not the current privilege level. When loading DS, ES, FS, and GS, it's OK to leave the RPL bits set to zero.

When a program is linked for execution in protected mode, the linker combines all of the program's code, data, and stack segments into a single large program segment. Because segmented protected-mode links are not currently supported, it is not possible to dynamically obtain segment selector values in a program. Instead, all segments initially allocated to a program have hardwired segment selector values.

When 386 | DOS-Extender loads the program, it creates two entries in the LDT that map this program segment. The descriptor accessed by segment selector value 000Ch maps the program segment as a readable code segment. This is the selector value initially loaded into the CS register. The descriptor accessed by segment selector value 0014h maps the program segment as a writable data segment. This is the selector value initially loaded into the DS, SS, ES, FS, and GS segment registers.

When a program first gains control, the registers contain the following values:

Level 3 values	Level 0 values
DS = 0017h	0014h
ES = 0017h	0014h
FS = 0017h	0014h
GS = 0017h	0014h
CS = 000Fh	000Ch
EIP = program entry point	
SS = 0017h	0014h
ESP = initial top of stack	
EAX = 00000000h	
EBX = 00000000h	
ECX = 00000000h	
EDX = 00000000h	
ESI = 00000000h	
EDI = 00000000h	
EBP = 00000000h	

The only exceptions occur when the Weitek or Cyrix coprocessors are present in the system. When a Weitek floating point coprocessor is present, a descriptor accessed by segment selector value 003Ch is set up to map the Weitek address space. The FS register is initialized to contain selector 003Ch for level 0, 003Fh for level 3.

If a Cyrix EMC87 coprocessor is present, a descriptor accessed by segment selector value 0048h is set up to map the EMC87's address space. No segment register is loaded with this selector; the segment register used to access the EMC87 is selected by the compiler runtime library.

3.2 Program Segments

When a protected-mode program is loaded into memory, several segments with hardwired segment selector values are set up in the LDT for use by the program. Table 3-1 lists the segments set up in the LDT when the program gains control. Separate columns in Table 3-1 show the segment selectors at level 0 and level 3; for level 3, the two LSBs are set.

TABLE 3-1
HARDWIRED LDT SEGMENTS

Segment Selector Value		
At Level 0	At Level 3	Description
0004h	0007h	A writable data segment that points to the MS-DOS program segment prefix (PSP) for the program.
000Ch	000Fh	A readable code segment that points to the program segment. This is the selector value initially loaded into the CS register.
0014h	0017h	A writable data segment that points to the program segment. This is the selector value initially loaded into the DS, SS, ES, FS, and GS registers.
001Ch	001Fh	A writable data segment that points to the physical memory for the screen. This is used by programs that wish to access directly the screen memory for performance reasons. The base address and limit of this segment are automatically updated by 386|DOS-Extender when the BIOS system call to change the video mode (INT 10h function 0) is made.
0024h	0027h	A writable data segment that is a duplicate of the descriptor accessed by selector value 0004h —i.e., it points to the program's PSP.
002Ch	002Fh	A writable data segment that points to the MS-DOS environment block for the program. The environment block contains a sequence of zero-terminated ASCII strings identifying the environment variables.

(cont.)

Segment Selector Value		
At Level 0	At Level 3	Description
0034h	0037h	A writable data segment that maps the entire first megabyte of memory used by MS-DOS.
003Ch	003Fh	A writable data segment that maps the memory space used by the Weitek 1167, 3167, or 4167 floating point coprocessor. If the Weitek is present in the system, this descriptor is initialized and the FS register is loaded with this segment selector value (003Ch, for level 0). If the Weitek is not present, the base and limit for this segment both are set to zero, and the FS register contains the same selector value as DS (0014h).
0044h	0047h	This selector is reserved to map the graphics memory on some Japanese machines that separate text and graphics memory, each with its own video buffer. On IBM PC and compatible machines, this selector maps the same memory as segment 001Ch.
0048h	004Fh	A writable data segment that maps the memory space used by the Cyrix EMC87 coprocessor. If the EMC87 is present in the system, this descriptor is initialized. If the EMC87 is not present, the base and limit for this segment both are set to zero. (At run time, a program can determine whether the Cyrix is present by calling the Get Configuration Information system call (function 2526h).)

In addition to these hardwired segments, new segments can be created in the LDT by making memory allocation system calls.

The organization of the hardwired LDT segments in the linear address space is shown in Figure 3-2.

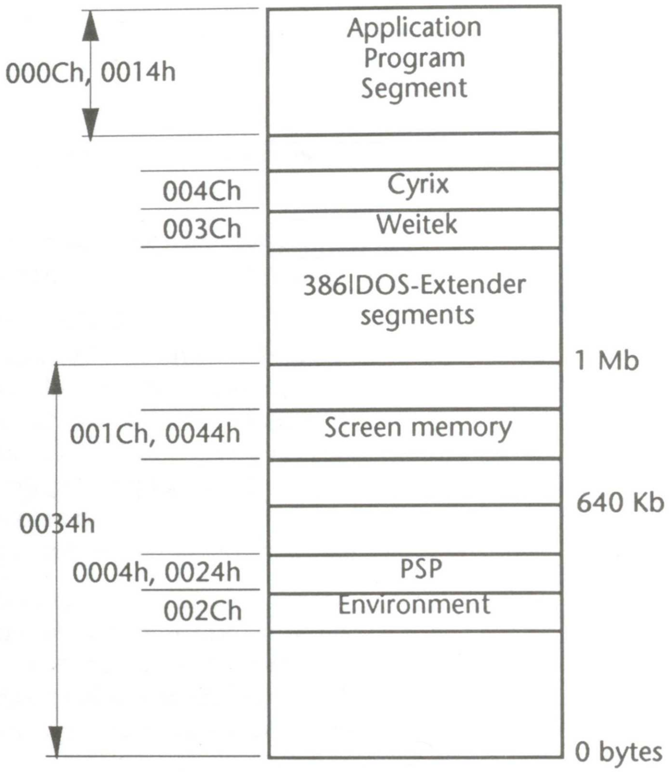


FIGURE 3-2
LDT SEGMENTS IN THE LINEAR ADDRESS SPACE

3.3 Program Segment Prefix (PSP)

Every program that runs under MS-DOS has a program segment prefix (PSP). This is a 256-byte block of memory that contains some program information maintained by MS-DOS. It also contains the command line that is passed to the program when it begins execution. Since the PSP must be accessible by MS-DOS, it always resides in conventional memory. Segment selectors 0004h and 0024h can be used to access the PSP while in protected mode.

The command line tail (the characters following the program name on the command line that invokes the program) is at offset 0081h in the PSP. The byte at offset 0080h in the PSP gives the length of the command tail string, not including the carriage return character that terminates the command tail.

The 128-byte buffer at offset 0080h in the PSP is also set up as the initial disk transfer area (DTA), which is used by the MS-DOS directory search system calls.

The first 128 bytes of the PSP contain information that is maintained by MS-DOS, including file handles for files opened by the program. Please see Reference 11 in the Preface for details on this portion of the PSP. In addition to the MS-DOS information, 386 | DOS-Extender places some information on the size of the program in the PSP. Table 3-2 identifies the information 386 | DOS-Extender stores in the PSP.

TABLE 3-2
386|DOS-EXTENDER PSP FIELDS

PSP Offset	Information
0000005Ch	The minimum size of the program segment, in bytes. This is equal to the actual number of bytes read in from the .EXP file, plus the **minimum** amount of extra data required to be allocated to the program (the minimum extra data is used for uninitialized data and stack memory), plus any base offset relocation value for the program (please see section 3.7).
00000060h	The allocated size of the program segment, in bytes. This gives the amount of memory actually allocated to the program when it was loaded. It is equal to the program segment's initial limit plus one byte.
00000064h	The total amount of extra memory, in bytes, allocated to the program segment when it was loaded. The allocated size of the program segment minus this value gives the amount of data that was read in from the .EXP file, plus the program's base relocation offset.

3.4 Environment Block

Every program running under MS-DOS has an environment block that contains a sequence of zero-terminated strings of the form "name=string." The entire block is terminated by an extra zero byte. Since the environment block is used by MS-DOS, it is always in conventional memory. Segment selector 002Ch can be used to access the environment block in protected mode. In addition, the real-mode paragraph address of the environment block is stored in the PSP at offset 002Eh.

The environment block contains strings used by the system's command interpreter (usually COMMAND.COM). It can also be used to pass information from a parent program to a child program. More information about the environment block is given in Reference 3 in the Preface.

3.5 Memory Allocation and Deallocation

Protected-mode programs have the ability to allocate and deallocate memory via the standard MS-DOS system calls. However, unlike most MS-DOS calls, 386 | DOS-Extender does not pass the memory allocation calls through to MS-DOS. Instead, it handles memory allocation itself, allocating and freeing memory in units of four-KB pages (rather than the MS-DOS units of 16-byte paragraphs). In addition, the "allocate memory" system call creates a segment descriptor entry in the LDT to map the allocated memory.

386 | DOS-Extender also provides the ability to make calls to the MS-DOS memory allocate and deallocate functions. These calls are supported to provide a way to allocate conventional memory. These calls are often needed by programs that call real-mode code, and are described in detail in Chapter 7.

Programs that allocate and deallocate memory via system calls must be aware of how memory is allocated to the program when it is loaded. When a program is linked, two parameters are initialized in the .EXP file, specifying the minimum and maximum amount of extra memory to allocate when the program is loaded. By default, the linker sets the minimum amount of extra data to allocate as the total amount of uninitialized data and stack memory used by the program. It also sets the maximum amount of extra data to allocate as all the memory in the machine.

The linker -MINDATA and -MAXDATA switches can be used to override these defaults. Well-behaved programs either should make a system call to release allocated memory they do not need, or should be linked with the -MAXDATA switch to prevent the loader from allocating all available memory to the program when it is loaded. If this is not done, it is impossible to allocate memory via system calls (or to EXEC another program), since the program already owns all the memory in the machine. For programs written in a high-level language, the initializer in the runtime library usually takes care of freeing up unused memory.

3.6 Aliased Segments

Aliased segments are separate segments that point to the same block of physical memory. For example, when an application program is loaded, a single program segment containing the code, data, and stack is created, and two aliased segments (code and data, selectors 000Ch and 0014h) are initialized to point to it. Aliased segments can also be created with the 2513h system call, and code and data segment aliases are automatically created when a program is loaded with the Load Program system call (2529h).

When the size of a segment with aliases is changed with the INT 21h function 4Ah call, the segment descriptors for all of the aliased segments are automatically updated.

However, when a segment with aliases is freed with a INT 21h function 49h call, only the segment descriptor is freed; memory allocated to the segment is not freed until the last of the aliased segments is freed.

3.7 Null Pointer Detection

One common programming error is attempting to reference memory through a null (zero) pointer. 386 | DOS-Extender can provide automatic detection for this error. When the program is linked, the -OFFSET switch can be used to place the beginning of the program one or more four-KB pages up from the base of the program segment. When 386 | DOS-Extender loads the program and initializes the program segment, it does not map the unused pages at the beginning of the segment (please see Chapter 5 for a discussion of paging). Then, any attempt to reference memory in the unmapped region of the program segment results in a page fault instead of corrupting the program's code or data.

This option does not waste any memory, because the missing pages are not mapped.

The -OFFSET switch is not supported under DPMI version 0.9 environments such as Windows 3.0 because it requires the DPMI Paging Support capability (please see Appendix F). 386 | DOS-Extender will refuse to run a program linked with -OFFSET under DPMI 0.9.

3.8 Performing an EXEC to Another Program

The standard MS-DOS system call (with extended parameters) is available for performing an EXEC to another program. When the child program terminates, control returns to the parent. The exact format of the EXEC system call is documented in Appendix A, function 4Bh.

In order to EXEC to a second program, there must be sufficient memory free to load the program. If the child program is a standard MS-DOS real-mode program, there must be enough conventional memory free to load the program. If the child program is a protected-mode program, there must be sufficient conventional memory available to load a second copy of 386 | DOS-Extender (approximately 65 KBs) and enough additional memory (either conventional or extended) available to load the program itself.

By default, 386 | DOS-Extender allocates all conventional memory in units of four-KB pages when the application program allocates memory. This default can be overridden by the use of the -MINREAL and -MAXREAL command line switches, either when the program is linked or when it is run with 386 | DOS-Extender. To EXEC to another program, you must either use these switches, or use system call 2525h to free some conventional memory before the EXEC.

When performing an EXEC to another protected-mode program, the amount of extended memory left free must also be considered, since protected-mode programs are capable of using extended as well as conventional memory. Extended memory is allocated dynamically when a program makes a memory allocate system call, as described in section 3.5. Thus, programs that EXEC to a second protected-mode program should only allocate memory they actually need, in order to leave additional memory free for the child program. System calls 2521h, 2525h, and 2536h enable you to restrict the use of physical extended or conventional memory. For more detailed information on memory allocation, please see Chapter 5; for specific system call data, please see Appendix B.

3.9 Environment Detection

386 | DOS-Extender is compatible with a variety of hardware and software environments, including DESQview 386, VCPI, and XMS. The same environment is always provided to the application, regardless of the actual machine configuration, so application programs can usually ignore the environment. For programs that need information about the environment, the Get Configuration Information system call (function 2526h) returns environment information.

3.10 Packed Programs

Protected-mode programs can be packed at link time by using the 386 | LINK -PACK command line switch. This option is used to reduce the size of the .EXP file on the disk and is particularly effective with FORTRAN programs, which typically contain a lot of zero data bytes. The -PACK option does not affect the size of the program in memory. Note that packed programs often have long load times under 386 | VMM, because the entire program must be read into virtual memory at program load time.

3.11 Simple Program Examples

To demonstrate the steps involved in developing a simple program to run in protected mode, this section lists a C program and an assembly language program, both of which print out the message "Hello world!" on the terminal when they are executed.

The program below is a C program that prints out "Hello world!":

```
#include <stdio.h>
main()
{
    printf("Hello world!\n");
    return 0;
}
```

This program is trivial; it just calls a C library function to print out the string. In fact (like most high level language programs) the same source code can be used to create either a real-mode or a protected-mode program, depending on how it is built. To create a protected-mode program, the source code file must be compiled with an 80386 C compiler to create an object file, and the object file must then be linked with the C library to create a program. If the source code file above is named CHELLO.C, then the following commands create a protected-mode program named CHELLO.EXP. This example uses the MetaWare High C compiler.

```
hc386 chello
386link chello -lib hcc
```

An assembly language program that also prints out the message "Hello world!" is listed below.

```
assume cs:cseg,ds:data

data     segment byte public use32 'data'
hellomsg db      'Hello world!',0Dh,0Ah,'$'
data     ends

cseg     segment byte public use32 'code'
print    proc    near
         lea     edx,hellomsg    ; call DOS to
         mov     ah,9     ; print string
         int 21h          ;

         mov     ah,4Ch   ; call DOS to
         int     21h      ; exit
print    endp
cseg     ends
```

```
sseg    segment byte public use32 'stack'
        db      1000h dup (?)
sseg    ends

end     print
```

This program is also very simple. It just calls MS-DOS to print out the string on the terminal. However, it does vary slightly from a real-mode program that performs the same function. A real-mode program would load the offset of the string into the DX register rather than the EDX register. This demonstrates two important characteristics of protected-mode assembly language programs: they tend to use 32-bit registers and operands as opposed to 16-bit operands in real mode, and system calls with pointers take a 32-bit offset rather than a 16-bit offset. If the above source code file is named AHELLO.ASM, then the following commands are used to assemble it to create an object file and to link it to create a protected-mode program named AHELLO.EXP:

```
386asm ahello
386link ahello
```

Chapter 4

Hardware Access

4.1 Privilege Levels

The 80386 processor allows programs to execute at a privilege level from zero to three, with zero the most privileged level. There is an I/O privilege level field (IOPL) maintained in the processor flags word (EFLAGS). When the current privilege level (CPL) is numerically greater (lower privilege) than the IOPL, the IN and OUT instructions used to access hardware I/O ports, and the enable and disable interrupts instructions, cause a general protection processor exception.

For applications executing at privilege level 3, 386 | DOS-Extender sets the IOPL in EFLAGS to 3 to let them do direct I/O.

In addition, instructions that load and store sensitive processor registers cause a general protection exception when executing at any privilege level other than zero. Privileged operations that are useful to applications, such as reading and writing the LDT, are supported with system calls in 386 | DOS-Extender. While applications can be run at privilege level 0 with the -PRIV switch, it is recommended that unprivileged operation (the -UNPRIV switch) be used.

Under DPMI, a host (such as Windows 3.0) can disallow direct I/O. However, applications can still contain such instructions; the host must virtualize them. When the instruction is executed, the host gets control through a processor exception, and gets the intended action done within the host environment. From the point of view of the application program, it appears as if the I/O instruction executed normally.

4.2 Hardware I/O

Hardware I/O in protected mode is performed in exactly the same way as in real mode — by using the IN and OUT instructions to read and write I/O ports. For memory-mapped hardware devices, system calls are available to create LDT segments that map physical memory anywhere in the four-gigabyte address space of the 80386 processor. Please see Appendix B for a list of 386 | DOS-Extender system calls. Memory-mapped devices in the first megabyte of memory can be accessed through segment selector 0034h, which maps the entire real-mode one-megabyte address space.

4.3 Screen Access

386 | DOS-Extender automatically sets up segment selector 001Ch to map the video refresh buffer for the video display adapter in the system. Programs that wish to read and write the display memory directly for performance reasons can do so through segment 001Ch. When a protected-mode program uses the BIOS system call INT 10h function 0 to change the video adapter mode, 386 | DOS-Extender automatically updates segment 001Ch to map the correct video memory.

4.4 Programming the Intel 80287/80387 Floating Point Coprocessors

The presence of either the Intel 80287 or 80387 floating point coprocessor can be detected with the BIOS equipment check system call (please see Appendix C). To determine whether an 80287 or an 80387 is present, use the Get Configuration Information system call (function 2526h).

From the point of view of an application program, the only difference between an 80287 and an 80387 is that the 80387 has a few additional instructions. However, there is a difference important to handler routines for coprocessor exceptions. The FSAVE, FRSTOR, FSTENV, and FLDENV instructions all have as an operand a memory block containing a copy of the coprocessor environment. The format in which this environment is stored depends on (1) the current mode of the 80386 (real or protected), and (2) the operand size attribute in effect for the instruction (16-bit or 32-bit). Please see

Reference 2 in the Preface for a complete description of all possible environment block formats and the differences between the 80287 and the 80387.

On 80286 systems with an 80287 coprocessor chip, the format of the environment block depends not on the current mode of the 80286, but on the current mode of the 80287, which can also be placed in either real or protected mode! On 80386 systems, however, the operating mode of the 80287 is irrelevant at all times, and there is no protected mode for the 80387. Instead, everything is determined by the operating mode of the 80386 when the coprocessor instruction is executed.

When assembling code containing the FSAVE, FRSTOR, FSTENV, or FLDENV instructions (or any other 80386 or 80387 instructions), the operand size attribute for an instruction is determined by the following rules:

- ☛ If the use type of the current segment is USE16, the default operand size is 16-bit. If the use type is USE32, the default operand size is 32-bit. By definition, code executing in real mode is executing in a USE16 segment. The protected-mode program segment set up by 386|DOS-Extender is always a USE32 segment.
- ☛ If an operand size override prefix byte (66h) precedes the instruction opcode, the operand size attribute in effect for the instruction is toggled.

Thus, either operand size attribute can be used in both real- and protected-mode code. Unless something unusual is done, the operand size attribute will be 16-bit in real-mode code and 32-bit in protected-mode code. Please see the *386 | ASM Reference Manual* for more information on operand size override bytes and the conditions under which the assembler generates them.

There is one problem to be aware of when writing protected-mode programs which use the 80387 floating point coprocessor. 80386 chips manufactured prior to step D0 have a bug which can cause them to hang when communicating with an 80387 coprocessor. The -NOPAGE and -ERRATA17 switches provide software workarounds for this chip bug. Please see section 2.10 for details.

4.5 Programming the Weitek Floating Point Coprocessor

If a Weitek floating point coprocessor chip is present on the system, 386 | DOS-Extender takes two actions. It sets up segment 003Ch to map the memory address space used to program the Weitek coprocessor, and it initializes the FS segment register to contain segment selector 003Ch.

If the Weitek coprocessor is not present, 386 | DOS-Extender sets the segment base and limit values in the LDT descriptor for segment 003Ch to zero. FS is then initialized to the same value as DS (segment selector 0014h, the program segment). The application program can, therefore, detect the presence of the Weitek coprocessor by comparing the values in the DS and FS segment registers.

Some existing 80386 systems support the Weitek chip but do not reliably report its presence. 386 | DOS-Extender therefore has a command line switch for Weitek coprocessor detection. The switch -WEITEK ON instructs 386 | DOS-Extender to presume the Weitek coprocessor is present; -WEITEK OFF means to presume the Weitek coprocessor is not present. The switch -WEITEK AUTO instructs 386 | DOS-Extender to execute the Weitek-approved presence detection code, which is a call to a BIOS equipment check routine. This routine is supposed to report Weitek presence correctly on all machines that support the Weitek coprocessor, but is not always reliable. The default if no command line switch is given is -WEITEK AUTO.

386 | DOS-Extender does not initialize the Weitek chip; this is the responsibility of the application program or the compiler runtime initializer. An approved sequence for coprocessor initialization, as well as for the programming interface to the chip, is specified in Reference 10 in the Preface.

4.6 Programming the Cyrix EMC87 Floating Point Coprocessor

The Cyrix coprocessor chip can execute normal 80387 instructions, and can also perform higher performance floating point operations if a memory-mapped interface is used. If a Cyrix floating point coprocessor chip is present on the system, 386 | DOS-Extender sets up segment 004Ch to map the memory address space used to program that coprocessor.

If the Cyrix coprocessor is not present, 386 | DOS-Extender sets the segment base and limit values in the LDT descriptor for segment 004Ch to zero. The presence of the Cyrix coprocessor can be determined with the Get Configuration Information system call (2526h).

386 | DOS-Extender has a command line switch for Cyrix coprocessor detection, but it should not be necessary since the chip responds in a unique way that signals its presence.

The switch -CYRIX ON instructs 386 | DOS-Extender to presume the Cyrix is present; -CYRIX OFF means to presume the Cyrix is not present. The switch -CYRIX AUTO instructs 386 | DOS-Extender to execute the Cyrix-approved presence detection code. The default if no command line switch is given is -CYRIX AUTO.

386 | DOS-Extender does not initialize the Cyrix chip; this is the responsibility of the application program or compiler runtime initializer. An approved sequence for coprocessor initialization, as well as for the programming interface to the chip, is specified in Reference 9 in the Preface.

Chapter 5

Memory Management

5.1 386 | DOS-Extender Memory Model

386 | DOS-Extender uses the 80386 and 80486 processors' paging and segmentation capabilities to perform its memory management.

Protected-mode segments are referenced by the values loaded into segment registers. These values are called segment selectors; they reference a descriptor entry in either the GDT or the LDT (please see Chapter 3). The descriptor entry contains the 32-bit linear base address of the segment and the segment limit. Attempts to reference data beyond the segment limit result in a processor exception.

The 80386 paging hardware translates 32-bit linear addresses into the 32-bit physical addresses that are placed on the address bus. A linear address contains 20 bits that select a physical page of memory, and 12 bits giving an offset within that selected memory page. A physical memory page is therefore four KB in size. For a complete description of both paging and segmentation on the 80386, please see Reference 1 in the Preface.

A memory reference on the 80386 always includes a segment selector value and an offset within the segment. (The segment selector is either implied or directly specified in the instruction.) When a memory reference is made in protected mode, the processor constructs a physical memory address in the following manner:

1. It gets the linear base address of the referenced segment from the segment descriptor in the GDT or the LDT.

2. It adds the offset within the segment to the segment base address to create a linear address in the four GB address space of the 80386.

3. It uses the paging built into the processor to convert the linear address to the physical memory address that is put on the address bus.

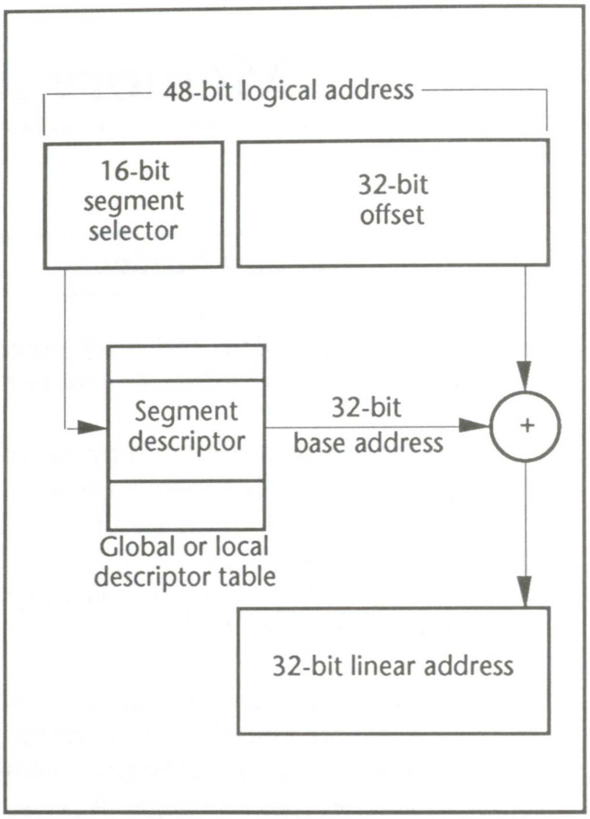


FIGURE 5-1
SEGMENT TRANSLATION

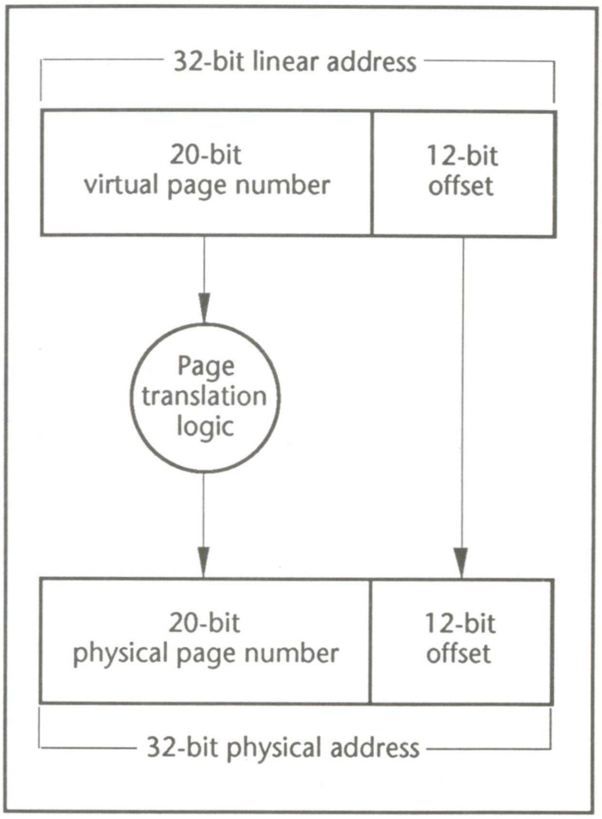


FIGURE 5-2
SIMPLIFIED PAGE TRANSLATION

This entire process can usually be ignored by protected-mode application programs. This chapter is provided as a reference for programs that need to use or manipulate the system descriptor tables or page tables.

5.1.1 Paging

Paging is controlled on the 80386 through page tables set up by 386 | DOS-Extender. A 32-bit linear address in protected mode is divided into three parts that use the page tables to point to a specific memory location.

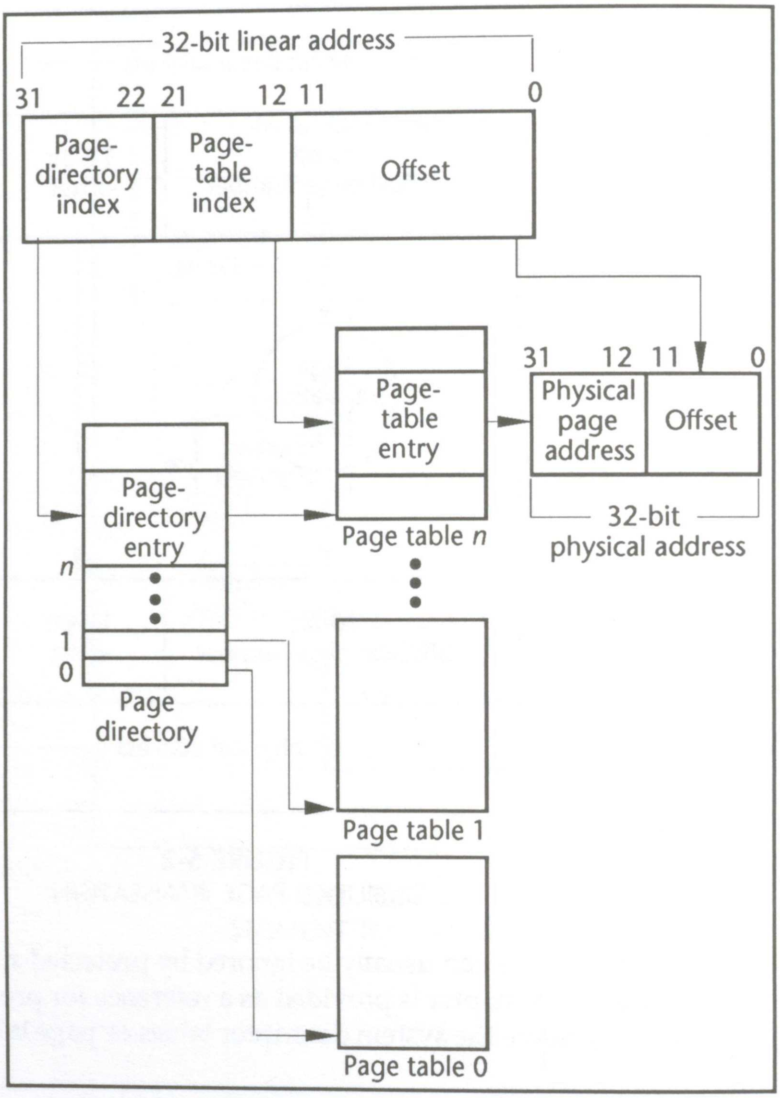


FIGURE 5-3
PAGE TRANSLATION

The high 10 bits in the linear address select an entry in the page directory, which points to a specific page table. The next 10 bits in the linear address select an entry in that page table, pointing to a specific physical memory page. The low 12 bits in the linear address is an offset within the selected physical memory page. This process is described in more detail in Reference 1 in the Preface.

The system contains only one page table called the page directory. Its physical address is held in a 80386 system register, CR3. Each entry in the page directory maps a physical memory page that is used as a page table. There can, therefore, be up to 1024 page tables in a system. Each entry in a page table maps a physical memory page.

All page tables are four KB in size, consisting of 1,024 four-byte entries. Each entry in a page table contains a 20-bit page frame address, mapping a single physical page, and 12 bits of flags. (Since each physical page is four KB in size, a 20-bit address is sufficient to address any page in the entire four-gigabyte address space.)

When 386 | DOS-Extender allocates memory, it allocates and maps one physical page at a time. Therefore, physical pages in a segment will not necessarily be contiguous.

The -NOPAGE command line switch can be used to disable paging. When this switch is used, the 386 | DOS-Extender normal memory management is also disabled. The protected-mode program is loaded entirely in extended memory, and the dynamic memory allocation system calls are disabled. This switch is provided as a workaround for a bug in the 80386 chip that occurs when the 80387 numeric coprocessor is being used. Please see section 2.10 for details.

5.1.2 Local Descriptor Table Segments

Entries in the local descriptor table (LDT) represent the segments owned by the application program. Table 3-1 gives a complete list of the LDT segments set up when the program is initially loaded. 386 | DOS-Extender creates new segments in the LDT whenever memory allocation requests are made with the INT 21h function 48h system call. Figure 5-4 shows the relationship between the LDT, the linear address space, and the physical address space.

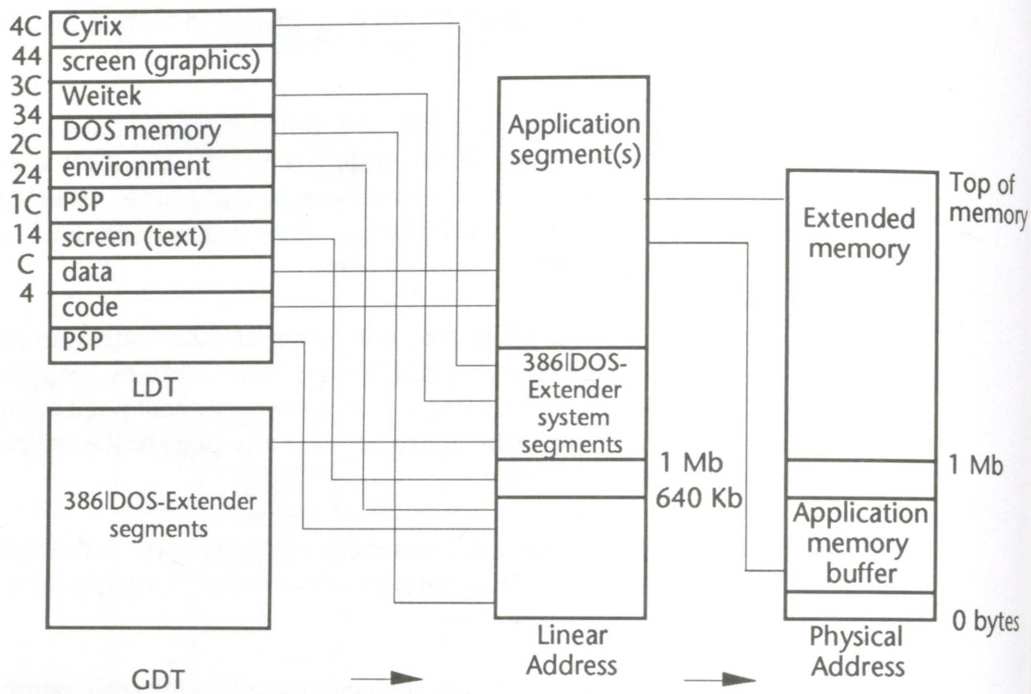


FIGURE 5-4
ADDRESS TRANSLATION

5.1.3 Global Descriptor Table Segments

Entries in the global descriptor table (GDT) represent the segments owned by 386 | DOS-Extender. These segments normally should not be used by the application program. (Please see Appendix H for information on using GDT segments at privilege level 0.)

5.2 Memory Allocation

386 | DOS-Extender always allocates and deallocates memory in units of pages (four KB). Both conventional memory (memory below one MB) and extended memory (memory above one MB) are available for use by the application program. When allocating memory to the application program, conventional memory is allocated first, and extended memory second.

Conventional memory is guaranteed to be allocated contiguously only when it is first allocated. This has some implications for writing programs that mix real-mode and protected-mode code (please see Chapter 7 for details). Of course, once the program frees up memory that it doesn't need, any subsequent memory allocation calls may get either conventional or extended memory, in no particular order.

Extended memory can be allocated from four distinct sources:

- Direct extended memory: This is also called the BIOS method, since it uses the BIOS call INT 15 function 88 (Get Size of Extended Memory). Programs can take over this call and reduce the reported size, thereby retaining control over the difference. (Please see section 5.2.2.)

- VCPI memory: VCPI (Virtual Control Program Interface) is an interface provided by most EMS emulators. EMS stands for Lotus/Intel/Microsoft Expanded Memory Specification. EMS emulators seize a block of memory at system boot time. The VCPI interface allows 386|DOS-Extender to dynamically allocate pages from this block of memory.

- XMS memory: Memory allocated using HIMEM.SYS, the memory allocator for extended memory from Microsoft. It is used in Windows 3.0 and DOS 5.0.

- "Special" memory: Built-in shadow memory such as in certain COMPAQ machines. This is not generally relevant, being available only in specific models of certain machines.

386 | DOS-Extender supports all but one combination of the above methods. The single exception arises from an ambiguity in the XMS specification. Consequently, if VCPI and XMS coexist, 386 | DOS-Extender will not try to get the XMS memory, but will use VCPI instead.

5.2.1 Conventional Memory Usage

Conventional memory is allocated out of a block of memory (the "application memory buffer") that 386 | DOS-Extender obtains by calling MS-DOS during initialization. The size of this block can be modified at run time with the 2525h or 2536h system calls.

By default, 386 | DOS-Extender uses the available conventional memory for the application memory buffer. The -MINREAL and -MAXREAL command line switches can be used either at link time or at run time to force 386 | DOS-Extender to leave some conventional memory free. This is necessary if the program wishes to perform an EXEC to a child program. Sufficient conventional memory must be left free to allow the child program to be loaded into memory.

By default, 386 | DOS-Extender uses approximately 80 KB of conventional memory for its own purposes. The minimum conventional memory requirements for 386 | DOS-Extender are approximately 65 KB (if -MAXIBUF 1 is used instead of the default setting of 16). The conventional memory usage of 386 | DOS-Extender is affected by the -MINIBUF, -MAXIBUF, -NISTACK, -ISTKSIZE, and -CALLBUFS command line switches. If these switches are used to allocate larger conventional memory data buffers than the defaults selected by 386 | DOS-Extender, the 386 | DOS-Extender use of conventional memory will be increased.

The following figure illustrates the typical organization of conventional memory:

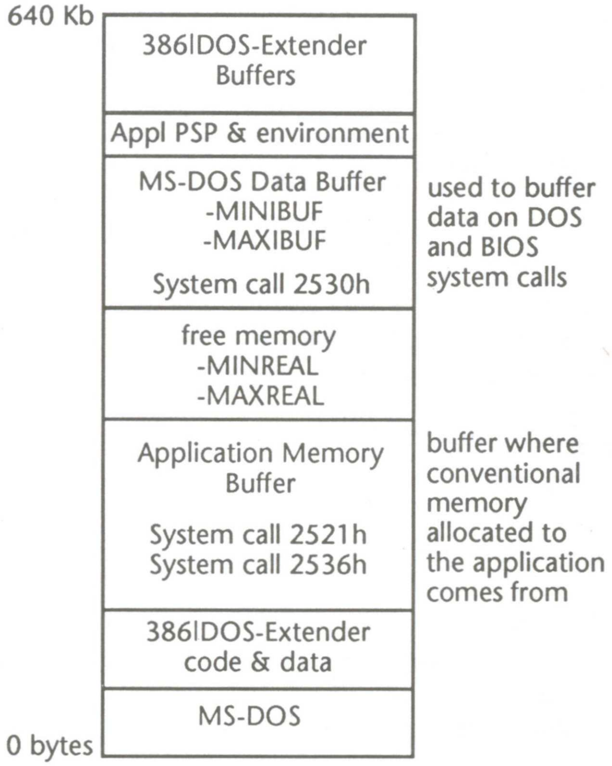

640 Kb
386|DOS-Extender Buffers
Appl PSP & environment
MS-DOS Data Buffer -MINIBUF -MAXIBUF System call 2530h
used to buffer data on DOS and BIOS system calls
free memory -MINREAL -MAXREAL
Application Memory Buffer System call 2521h System call 2536h
buffer where conventional memory allocated to the application comes from
386|DOS-Extender code & data
MS-DOS
0 bytes

FIGURE 5-5
CONVENTIONAL MEMORY USAGE

The sizes of the MS-DOS data buffer and the application memory buffer can be adjusted to make room for programs to be EXEC'd. The load-time switches that affect the MS-DOS Data Buffer are -MINIBUF and -MAXIBUF; those that affect the area of free memory (and therefore indirectly the size of the application memory buffer) are -MINREAL and -MAXREAL.

At run time, the Set DOS Data Buffer Size system call (2530h) can be used to adjust the size of the MS-DOS data buffer, and either Limit Program's Conventional Memory Usage (2525h) or Minimize/Maximize Extended/Conventional Memory Usage (2536h) can be used to adjust the size of the application memory buffer.

System call 2537h, Allocate Conventional Memory Above DOS Data Buffer, enables an application program to allocate conventional memory and also to retain the ability to resize both the MS-DOS data buffer and the application memory buffer. (This flexible combination is not possible using MS-DOS INT 21h function 25C0h. Please see call descriptions in Appendix B.)

5.2.2 Direct Extended Memory Usage

During initialization, 386 | DOS-Extender determines how much direct extended memory is not in use by other programs. There are two de facto standards in place for allocating direct extended memory: the VDISK standard (the original IBM RAM disk program for the PC AT) and the BIOS extended memory determination call.

Programs that follow the VDISK standard allocate memory starting at one MB and growing upward. The memory allocation is marked both by a signature block maintained in conventional memory pointed to by the INT 19h interrupt vector, and in a boot block maintained at one MB.

If more than one program using this scheme is installed, the second and subsequent programs are required to increase the memory allocation mark in both of these locations. Programs that use the VDISK standard include RAM disk drivers. (Please see also section 8.9.1.)

Programs that use the BIOS method for allocating extended memory do so from the top down. The BIOS extended memory determination system call (INT 15h function 88h) returns the amount of extended memory in the system. The BIOS routine invoked by this function call returns the actual amount of physical memory in the system.

Programs using this method make the system call to determine how much memory is currently available. They then take over the function call and reduce the amount of available extended memory by the amount they take for their own use. Examples of such programs include disk cache drivers, EMS emulators, and XMS drivers. (Please see also section 8.9.2.)

386 | DOS-Extender, therefore, sizes direct extended memory at initialization time by checking for VDISK allocation from one MB up, and by making the BIOS call to find the top of extended memory. If you have programs installed that don't use either of these standards, you can force

386 | DOS-Extender to skip the used memory locations with the -EXTLOW and -EXTHIGH command line switches.

386 | DOS-Extender allocates direct extended memory by taking over INT 15h function 88h. It allocates extended memory pages one at a time, from the top down, until there are no more available. Since extended memory is allocated incrementally, if a protected-mode program performs an EXEC to a child protected-mode program, the child program will have available to it all extended memory that was not used by its parent. When the child program terminates, all the memory allocated to it is automatically released and is again available for use by the parent program.

Programs typically take all of memory when they are first loaded (this is controlled with the -MINDATA and -MAXDATA linker switches). As part of program initialization, a well-behaved program then releases all the memory it does not need. 386 | DOS-Extender raises the top of extended memory allocation mark as extended memory pages are released, so that they will be available to programs subsequently EXEC'd.

Of course, given the primitive nature of the allocation scheme, pages released in the middle of allocated extended memory cannot be used to raise the INT 15h function 88h allocation marker. When this occurs, the freed memory will still be available for reallocation by the program that freed it, but will not be available to any child programs. The Limit Program Extended Memory Usage (2521h) and Minimize/Maximize Extended/Conventional Memory Usage (2536h) system calls can be used to free up physical extended memory pages for use by another program.

Chapter 6

Interrupts and Exceptions

Interrupts can be classified into three categories:

- hardware interrupts (caused by external hardware)
- software interrupts (caused by executing an INT instruction)
- processor exceptions (generated by the 80386 processor when it detects certain programming errors).

This chapter describes how these interrupts are handled in both protected-mode and real-mode environments. This chapter assumes you are somewhat familiar with writing real-mode interrupt handlers under MS-DOS.

Section 6.1 describes the standard 386 | DOS-Extender processing for interrupts that occur during protected-mode execution, and particular considerations that apply to MS-DOS system calls (made using software interrupt 21h).

Section 6.2 provides some background information on installing interrupt handlers under 386 | DOS-Extender.

Section 6.3 gives brief, descriptive overviews of the main system calls used for getting and setting interrupt vectors.

Section 6.4 discusses fundamental concepts used to write protected-mode or real-mode interrupt handlers.

Section 6.5 documents the interrupt stack frame presented to protected-mode interrupt handlers.

Section 6.6 outlines some typical strategies used to write protected-mode interrupt handlers. Several of these strategies are demonstrated in the sample interrupt handlers included on the product distribution disks and described in Appendix E.

Section 6.7 documents special considerations on the use of the hardware interrupt enable flag in a DPMI environment.

Section 6.8 specifies certain PC compatibility issues and describes simple workarounds to solve them.

6.1 386 | DOS-Extender Standard Interrupt Processing

This section describes the normal processing performed by 386 | DOS-Extender for interrupts occurring in protected mode.

Since applications can run at privilege level 3, application interrupt handlers cannot get control directly through the IDT. Instead, a 386 | DOS-Extender handler running at level 0 gets control first and then passes control to the application's handler at the appropriate privilege level.

Unless the application program has taken over the interrupt vector (please see section 6.3), protected-mode interrupts give control to 386 | DOS-Extender. It switches the processor to real mode and reissues the interrupt as a software interrupt, to be handled by the real-mode interrupt handler.

All general registers are preserved to the real-mode handler, as are the processor flags. The SS:SP registers are set to point to a stack memory buffer maintained internally by 386 | DOS-Extender. The size of the stack buffer defaults to 1 KB and is controlled by the -ISTKSIZE command line switch. The CS:IP registers will be within 386 | DOS-Extender code when the interrupt is reissued in real mode. The contents of other segment registers (DS, ES, FS, GS) are destroyed when the real-mode handler gets control.

When the real-mode interrupt handler completes, 386 | DOS-Extender switches to protected mode and returns control to the code that was executing when the interrupt occurred. The SS:ESP and CS:EIP registers are, of course, restored to what they were when the interrupt occurred in protected mode. The DS, ES, FS, and GS registers are also restored to their original values. All general registers and flags contain the values the real-mode interrupt handler left in them.

The total overhead required for 386 | DOS-Extender to switch from protected mode to real mode, or to switch from real mode to protected mode, is approximately 58 microseconds on a 25 MHz 80386 machine, and approximately 30 microseconds on a 25 MHz 80486 machine.

6.1.1 Software Interrupts

When a software interrupt occurs in protected mode, 386 | DOS-Extender switches the processor to real mode and reissues the interrupt. For MS-DOS or BIOS system calls, it first converts any 32-bit protected-mode parameters to the 16-bit real-mode format that DOS expects. For most such calls, the calling sequence is identical to that employed by real-mode programs. Appendixes A and C specify the 32-bit protected-mode format of MS-DOS and BIOS system calls, respectively.

386 | DOS-Extender system calls, like MS-DOS system calls, are invoked with software interrupts. They are made with interrupt 21h, with register AH set to 25h and the function code for the desired system call in register AL. Appendix B gives a detailed description of each 386 | DOS-Extender system call.

System calls without pointer parameters are usually passed through to real mode unmodified, that is, the standard interrupt processing described in section 6.1 is performed.

System calls with pointers must be modified because the contents of segment registers are different in protected mode and real mode. (Also, a protected-mode data buffer can reside above 1 MB in physical memory, where it cannot be addressed in real mode. Protected-mode segments can be up to 4 GB in size, while real-mode segments are limited to 64 KB.)

In protected mode, segment registers contain an index to an entry in the GDT or LDT.

In real mode, segment registers contain a memory paragraph address.

When a protected-mode system call includes a pointer, 386 | DOS-Extender normally buffers the data in a conventional memory buffer. MS-DOS or the BIOS receives a pointer to the conventional memory buffer rather than the application program's buffer. The -DATATHRESHOLD switch can be used to force 386 | DOS-Extender to convert the application's pointer to a real-mode pointer instead of buffering the data. Pointer conversion is possible if the application's buffer resides in contiguous conventional memory. Usually, this is not the case, and the check for whether pointer conversion is possible requires almost as much overhead as buffering data, so better overall performance is obtained by always buffering data. The primary reason for using -DATATHRESHOLD is for re-entrant capabilities in multitasking applications; please see Appendix K for details.

The internal data buffer for system calls varies in size from 1 to 64 KB. It is controlled by the -MINIBUF and -MAXIBUF command line switches, and the Set DOS Data Buffer Size system call (function 2530h). To transfer data in amounts larger than the internal buffer, 386 | DOS-Extender makes repeated calls to DOS or the BIOS.

When the system call returns, the original segment register values are usually restored, as described for the standard interrupt processing. However, when a pointer is returned, 386 | DOS-Extender converts it to a protected-mode pointer. This conversion usually uses segment 0034h, which maps the entire first MB of MS-DOS memory.

The above processing applies only to the documented MS-DOS system calls and the standard BIOS system calls. All undocumented system calls with pointers, or system calls for other programs that use pointers, such as network drivers, must be handled by the application program.

386 | DOS-Extender provides a system call (2511h) for issuing an arbitrary real-mode interrupt with all real-mode register values specified, **including** the segment registers. This system call also returns the real-mode values left in the segment registers by the real-mode interrupt handler. This 386 | DOS-Extender system call can be used to make any real-mode system call that is implemented with software interrupts. Chapter 7 discusses the more general problem of how to write protected-mode programs that communicate with real-mode programs.

6.1.2 Hardware Interrupts

Hardware interrupts normally occur at their standard MS-DOS locations: interrupt vectors 08h-0Fh for IRQ0-7, and 70h-77h for IRQ8-15. Even in environments where hardware interrupts occur on nonstandard interrupt vectors (please see Appendix G), 386 | DOS-Extender makes it appear as though hardware interrupts occur on their standard vectors.

Interrupt vectors 08h-0Fh are also used for 80386 processor exceptions. 386 | DOS-Extender can always determine whether an interrupt in this range is the result of a hardware interrupt or a processor exception. When you install a handler in the 08h-0Fh range, you specify whether the handler is for the hardware interrupt or the processor exception (please see section 6.3).

Although not recommended, 386 | DOS-Extender can be forced to relocate hardware interrupts IRQ0-7 so they do not conflict with processor exceptions. Please see Appendix L for details.

386 | DOS-Extender always performs the standard interrupt processing described in section 6.1 for hardware interrupts. Note that this processing allows the real-mode interrupt handler to modify general registers and processor flags, which a hardware interrupt handler must never do. 386 | DOS-Extender, therefore, allows incorrectly written hardware interrupt handlers to crash the machine, just as is possible in real mode.

6.1.3 Processor Exceptions

Except for the debug exceptions (1 and 3), any protected-mode processor exception causes 386 | DOS-Extender to terminate the program. It then prints a message identifying the exception and the address of the instruction that caused it. The same is true for real-mode exceptions 0, 4, 5, 6, and 7.

When this occurs, the program should be run under the Phar Lap 386 | DEBUG or 386 | SRCBug debugger. The debugger will gain control when the exception occurs and allow you to locate its cause by examining registers and memory locations.

6.1.4 MS-DOS Considerations

Under 386 | DOS-Extender, protected-mode programs use software interrupts to make MS-DOS system calls, exactly as real-mode programs do. They are handled as described above for software interrupts.

The file control block (FCB) file I/O calls are not supported under 386 | DOS-Extender. The handle-based file I/O calls must be used instead.

The MS-DOS get and set interrupt vector functions (25h and 35h) are not supported. They have been replaced by Phar Lap system calls that deal with the additional complexities of having two modes of processor operation.

Many undocumented MS-DOS system calls are in common use by programmers. Such system calls are passed through with no registers modified. This allows all functions, except those requiring values to be passed in segment registers, to work as they do in real mode.

A detailed description of each MS-DOS system call is given in Appendix A.

6.2 Taking Over Interrupts

Taking over interrupts is more complex for protected-mode programs than for real-mode programs. A protected-mode program taking over interrupts must know the processor mode when an interrupt occurs and be prepared to handle interrupts arising in either mode:

- Software interrupts, of course, occur only when some program issues them, so if you know what code might use a particular software interrupt, then you know in which mode the processor will be executing when the interrupt is issued.
- Hardware interrupts can occur at any time. A program taking responsibility for them must handle both those that occur in real mode and those that occur in protected mode.
- Processor exceptions, usually indicating program bugs, can occur in both real and protected mode and normally are not taken over by an application program. Pure protected-mode programs can only cause exceptions in protected mode. Programs with some code executing in real mode and some in protected mode can, of course, cause exceptions in either mode.

In protected mode, an interrupt invokes a handler through the interrupt descriptor table (IDT), which can reside anywhere in memory. It contains an 8-byte descriptor for each of the 256 possible interrupts. Each descriptor contains the entry point (segment selector and offset) of the interrupt handler routine, plus a code identifying the type of descriptor.

Although the 80386 processor permits using task, interrupt, or trap gates in the IDT, 386 | DOS-Extender always uses an interrupt gate. This means the handler is vectored to directly, without a task switch, and interrupts are always disabled when the handler gets control.

In protected mode, the hardware IDT entries always point to an "umbrella" handler in 386 | DOS-Extender. This umbrella handler invokes the actual interrupt handler (either the default 386 | DOS-Extender handler, or a handler installed by the application) through a "shadow" IDT. This shadow IDT is what gets read and written by the get and set interrupt vector calls. There is no system call that affords access to the hardware level protected-mode IDT. However, applications that choose to run at privilege level 0 (please see Appendix H) can read and write the IDT directly with a GDT segment.

For each protected-mode interrupt from 08h to 0Fh, the shadow IDT has two entries that can be read and written via separate system calls. One is for a hardware interrupt handler and the other is for a processor exception handler. The umbrella 386 | DOS-Extender handler for interrupts 08-0F determines whether the source of the interrupt is a hardware interrupt or a processor exception. For hardware interrupts, control transfers to the hardware interrupt handler; otherwise, control transfers to the processor exception handler.

For hardware and software interrupts, the default 386 | DOS-Extender protected-mode interrupt handlers switch to real mode and reissue the interrupt. For processor exceptions, they terminate the application with an error message.

In real mode, when an interrupt occurs, the processor vectors through an address table that can reside anywhere in the first 1 MB of memory. Under MS-DOS, it is always located at 00000h - 003FFh. Each table entry contains the 4-byte real-mode segment:offset address of the interrupt handler to be invoked. Interrupts are always disabled when real-mode interrupt handlers get control.

In real mode, 386 | DOS-Extender installs handlers only for processor exceptions 0 and 4-7, and for the DOS CTRL-C interrupt (23h). If any of these interrupts occur, the application is terminated with an error message.

6.3 Get and Set Interrupt Vector Calls

System calls 2502h-2507h are used to get and set interrupt vectors for all kinds of interrupts (processor exceptions, hardware interrupts, and software interrupts) **except** processor exceptions in the 08h-0Fh range.

For protected-mode interrupts 08h - 0Fh, there are two possible sources:

☛ For hardware interrupts, use 2502h to get the current vector, and 2504h or 2506h to set the vector.

☛ For processor exceptions, use 2532h to get the current vector, and 2533h to set the vector.

For real-mode interrupts 08h-0Fh, 386 | DOS-Extender does not distinguish between processor exceptions and hardware interrupts. If you install a real-mode handler in this range, it will get control if either a hardware interrupt or a processor exception occurs.

For protected-mode handlers, the original 386 | DOS-Extender handler in the shadow IDT always runs at the same privilege level as the application. This has the following consequences:

☛ Regardless of privilege level, the application handler can chain to the previous 386|DOS-Extender handler.

☛ The application handler can also set up a forged interrupt stack before chaining, in order to regain control after the original 386|DOS-Extender handler completes.

6.3.1 Get Protected-Mode Interrupt Vector

This system call (function 2502h) is used to save the original value of a protected-mode interrupt vector in the shadow IDT before modifying it. This enables the program to restore that original vector at a later time.

This call should not be used for processor exceptions 08h - 0Fh; use 2532h instead.

6.3.2 Get Real-Mode Interrupt Vector

This system call (function 2503h) returns the real-mode segment address in the high 16 bits of EBX and the offset in the low 16 bits of EBX.

It is used to save the original value of a real-mode interrupt vector in the real-mode interrupt table before modifying it. This enables the program to restore that original vector at a later time.

6.3.3 Set Protected-Mode Interrupt Vector

This system call (function 2504h) modifies the shadow IDT, creating an entry that points to a protected-mode interrupt handler. This function is useful for trapping software interrupts and processor exceptions that the programmer knows will occur only in protected mode.

This call should not be used for exceptions 08h - 0Fh; use 2533h instead.

6.3.4 Set Real-Mode Interrupt Vector

This system call (function 2505h) modifies a pointer in the real-mode interrupt vector table, creating an entry that points to a real-mode interrupt handler.

386 | DOS-Extender always responds to a protected-mode interrupt by switching to real mode and reissuing the interrupt. Therefore, taking over a real-mode interrupt vector means your real-mode handler will get control regardless of the mode in which the interrupt occurs.

This function is useful for taking over hardware interrupts so as to process them in real mode. It is also used for taking over real-mode software interrupts that may be easier to process in real mode, such as the MS-DOS critical error interrupt.

6.3.5 Set Interrupt to Gain Control in Protected Mode

This system call (function 2506h) makes two changes:

- it modifies the protected-mode shadow IDT, creating an entry that points to a protected-mode interrupt handler
- it modifies the real-mode interrupt vector table to point to a 386|DOS-Extender routine that switches to protected mode and reissues the interrupt.

Therefore, this system call causes the protected-mode interrupt handler to gain control, regardless of the mode in which the interrupt originally occurs.

This function is useful for taking over hardware interrupts for processing in protected mode. It is also used for taking over real-mode software interrupts that the programmer wants to process in protected mode, such as the DOS CTRL-C interrupt. Section 6.4 describes the environment provided to an interrupt handler that gets control by a switch from real mode to protected mode.

This call should not be used for exceptions 08h-0Fh. To take over exceptions 08h-0Fh in protected mode only, use function 2533h .

In real mode, 386 | DOS-Extender always considers interrupts 08h-0Fh to be hardware interrupts. If you need to distinguish between processor exceptions and hardware interrupts in real mode, you must write a real-mode handler that can determine the source of the interrupt, and install it with function 2505h.

6.3.6 Set Real- and Protected-Mode Interrupt Vectors

This system call (function 2507h) has three effects:

- it modifies the protected-mode shadow IDT to set up a protected-mode interrupt handler
- it modifies the real-mode interrupt vector table to set up a real-mode interrupt handler
- it sets both vectors indivisibly, that is, interrupts are disabled until both vectors have been modified.

This third effect is the only difference between using this function and making separate calls to function 2504h and 2505h. It can be important, for example, when taking over a hardware interrupt with separate real-mode and protected-mode handlers.

6.3.7 Get Protected-Mode Processor Exception Vector

This system call (function 2532h) is used to obtain the protected-mode interrupt vectors for exceptions 00h-0Fh. The protected-mode interrupt vectors for exceptions 00h-07h may also be obtained with function 2502h.

6.3.8 Set Protected-Mode Processor Exception Vector

This system call (function 2533h) is used to install a protected-mode interrupt handler for exceptions 00h-0Fh. The protected-mode interrupt vectors for exceptions 00h-07h may also be set with functions 2504h and 2506h.

6.4 Fundamentals of Writing Interrupt Handlers

When an interrupt occurs, the processor pushes the flags and the current CS:(E)IP instruction address onto the stack. Then it transfers control to the interrupt handler.

The interrupt handler saves any additional registers it is going to modify, performs its processing, and restores all the saved registers. It then executes a "return from interrupt" instruction, which restores the saved flags and instruction address from the stack.

It is extremely important to select the correct interrupt return instruction, of which there are two forms:

- a 16-bit form, the IRET instruction mnemonic
- a 32-bit form, assembled by the IRETD instruction mnemonic.

When a protected-mode interrupt occurs, the processor **always** saves information on the stack in 32-bit form. Therefore, a protected-mode interrupt handler must **always** return with the IRETD instruction.

When a real-mode interrupt occurs, the processor **always** saves information on the stack in 16-bit form. Therefore, a real-mode interrupt handler must **always** return with the IRET instruction.

For more information on what the processor does when an interrupt occurs, please see Reference 1 in the Preface.

The subsections below discuss the following issues of importance when writing both real-mode and protected-mode interrupt handlers: saving registers, high-level language interface support, and re-entrancy.

Section 6.5 documents the stack frame presented to protected-mode interrupt handlers by 386 | DOS-Extender.

Section 6.6 describe some typical strategies used when writing protected-mode interrupt handlers.

Appendix E contains code and explanations for several examples of interrupt handlers.

6.4.1 Saving Registers

Since hardware interrupts can occur at any time, hardware interrupt handlers must save and restore all registers that they modify. Software interrupt handlers are often used to implement system calls (e.g., MS-DOS calls), so they often store return parameters in registers. To modify processor flags, interrupt handlers must modify the saved flags on the stack, since these are restored by the interrupt return instruction.

Hardware interrupt handlers must make no assumptions about their environment, since a hardware interrupt can occur at any time. A real-mode handler that uses more than 100 bytes or so of stack space should switch stacks when it gets control. A protected-mode handler is given a 1 KB stack allocated by 386 | DOS-Extender. No assumptions of any kind should be made about any segment registers except CS.

It is particularly important to preserve registers correctly when modifying an existing real-mode interrupt handler to run in protected mode. This is equally true when writing a program that uses an existing real-mode hardware device driver.

A good example is the Microsoft mouse driver. Typically, the mouse driver is called to set up an interrupt handler. Its purpose is to gain control each time the mouse moves or any of its buttons is pressed. The driver will preserve all the 16-bit registers and then invoke the specified handler each time the mouse interrupt occurs. Therefore, standard 8086 programs using the mouse need not worry about saving registers.

However, this is **not** the case for 80386 programs using the mouse driver, because they can modify the 32-bit registers of the 80386. Such modification is likely if a protected-mode interrupt handler has been set up with 386 | DOS-Extender system function 2506h, as described above in section 6.3.5.

6.4.2 High-Level Language Interface Support

If the handler makes any calls to routines written in a high-level language, it must set up the environment that the high-level language expects. Most languages expect the following:

- DS and ES point to the data segment (segment selector 0014h, or 0017h at privilege level 3).
- SS points to the same segment as DS. This will **not** be the case when any interrupt handler gets control. Therefore, a handler that performs processing in a high-level language must always switch to a stack in segment 0014h (0017h at privilege level 3) before calling the high-level language routine.
- The direction flag in the EFLAGS register is clear.

6.4.3 Re-entrant Interrupt Handlers

Hardware interrupt handlers must either be re-entrant, or guard against being re-entered. If they are not written to be re-entrant, they must prevent it by taking one of the following actions:

- leaving interrupts disabled
- not issuing the end-of-interrupt (EOI) command to the 8259 interrupt controller until done processing the interrupt
- using a semaphore to control entrance to the handler.

Finally, hardware interrupt handlers must **not** make DOS or BIOS system calls, since neither MS-DOS nor the BIOS is re-entrant.

Please see Appendix K for more information on re-entrant code.

6.5 Interrupt Stack Frame for Protected-Mode Handlers

As noted in section 6.2, an umbrella handler in 386 | DOS-Extender always runs before control is passed to the application handler. This is necessary because the interrupt may occur when 386 | DOS-Extender level 0 code is executing. If the application is running at level 3, the privilege level transition must be done by software, since the hardware IDT explicitly forbids transitions to a less privileged level.

The umbrella handler always switches to an internal, locked stack of at least 1 KB (controlled by the -ISTKSIZE switch). Before transferring control to the handler in the application, it also sets up the following stack frame (all values are DWORDs):

	Offset	Name	Description
	38h	CR2	at time of original interrupt
	34h	INTN	number of the interrupt that occurred
	30h	DOSXFLAGS	as defined by the umbrella handler; please see below
	2Ch	GS	at time of original interrupt
	28h	FS	at time of original interrupt
	24h	DS	at time of original interrupt
	20h	ES	at time of original interrupt
	1Ch	SS	at time of original interrupt
	18h	ESP	at time of original interrupt
	14h	EFLAGS	at time of original interrupt
	10h	CS	at time of original interrupt
	0Ch	EIP	at time of original interrupt
	08h	EFLAGS	for internal 386 | DOS-Extender umbrella handler
	04h	CS	for internal 386 | DOS-Extender umbrella handler
SS:ESP—>	00h	EIP	for internal 386 | DOS-Extender umbrella handler

If the handler is for processor exceptions 08h or 0Ah-0Eh, there is also a processor error code pushed on the stack below the EIP for the internal 386 | DOS-Extender handler.

In many circumstances, an interrupt handler doesn't care what was executing when the interrupt occurred, and doesn't need to examine or modify segment registers. Such a handler can be written just as if it were vectored to directly through the IDT.

This works because the bottom of the stack frame looks just like a hardware-level interrupt stack frame, that is, EFLAGS, CS:EIP, and an error code if one is pushed by a processor exception. So an application interrupt handler can IRETD, or chain normally.

These entries — the umbrella handler's CS:EIP and EFLAGS — are always at the same privilege level as the application, and must **not** be changed before returning.

The original CS:EIP, EFLAGS, and SS:ESP represent what was executing when the original interrupt occurred. You **can** change these to return to a different location.

The ES, DS, FS, and GS stack entries are the register contents at the time of the original interrupt. When the handler gets control, the actual ES, DS, FS, and GS registers contain the following:

- Original values if the LOADSEG bit in DOSXFLAGS (described below) is clear.
- Zero (the null selector) if the interrupted code was executing at level 0 and the application is at level 3 (the LOADSEG bit in DOSXFLAGS is set to one to indicate this condition). This enables the application to save the segment registers and later restore them. Otherwise, when a level 3 handler tried to save and restore the contents of a segment register, the register would contain a level 0 selector. This would cause the handler to get an exception on the restore operation.

Thus, when a handler gets control, the original segment register values are always available on the stack, and the registers themselves may or may not have their original values, depending on the setting of the LOADSEG bit. When the handler returns, the 386 | DOS-Extender umbrella handler reloads the segment registers with the values on the stack if the LOADSEG bit is set.

The bits in the DOSXFLAGS on the stack are defined as follows when the handler is invoked:

Name	Bit	Description
RMODE	00000001h	0, if interrupt originated in protected mode 1, if interrupt originated in real mode
LOADSEG	00000002h	0, if DS, ES, FS, and GS all contain their original values (equal to the values saved on the stack) 1, if DS, ES, FS, or GS have been set to zero (because the interrupt occurred in real mode at level 0 and the application runs at level 3)
	FFFFFFFCh	Reserved; always zero

When the handler returns, the DOSXFLAGS bits have the following meaning:

Name	Bit	Description
RMODE	00000001h	Not used
LOADSEG	00000002h	0, if do not modify DS, ES, FS, GS registers 1, if reload DS, ES, FS, GS with values in the stack frame
	FFFFFFFCh	Reserved; should not be modified

The application handler is permitted to set the LOADSEG bit, but is not permitted to clear it. If the handler returns with the segment register bit set, the umbrella handler always restores DS, ES, FS, and GS from the values on the stack. If the LOADSEG bit is clear, the values in the registers are not modified.

The INTN interrupt number is provided in the stack frame because the default 386 | DOS-Extender handlers use it to reissue the interrupt in real mode.

Saving the CR2 register on the stack provides a reliable way to get the CR2 value at the time a page fault occurs. The CR2 value on the stack is not guaranteed for any interrupt except the page fault.

Under 386 | VMM, an alternate page fault handler can itself cause page faults (which change the contents of CR2). It therefore needs a reliable way to get the appropriate CR2 value. Also, at level 3 the application has no way to read CR2 directly, because the instructions for reading or writing system registers are privileged instructions.

When the RMODE bit in DOSXFLAGS is set, it means the interrupt originally occurred in real mode. Because you installed your handler with the Set Interrupt to Always Gain Control in Protected Mode system call (function 2506h), 386 | DOS-Extender passes control directly to your protected-mode interrupt handler. In this case, the original register values saved in the interrupt stack frame are of course the values at the time the real-mode interrupt occurred.

Typically (for example, if your handler is for a hardware interrupt that can occur in real or protected mode), you won't care about the originating mode or the saved register values. However, if you wish, you can use these original register values and modify them when you return, just as you can with protected-mode interrupts.

For example, when MS-DOS issues a critical error (INT 24h) in real mode, it first saves information about the INT 21h that resulted in the critical error on the real-mode stack. Your handler could pick up the original real-mode SS:SP from the interrupt stack frame, and convert it into an offset in segment 0034h (which maps the MS-DOS memory space) to look at the real-mode interrupt stack. For an example of this, please see the discussion of the CRITERR interrupt handler in Appendix E.

6.6 Typical Strategies for Protected-Mode Handlers

Most protected-mode interrupt handlers perform one or more of the following operations:

- process the interrupt and then either IRETD or chain to the previous handler
- change registers when you return from the interrupt

- chain to the previous handler but regain control when it returns
- retain control and never return from the interrupt.

The following subsections describe how to accomplish these goals. Please see also the sample interrupt handlers described in Appendix E.

6.6.1 To Process an Interrupt and Then IRETD

To process an interrupt and then return to the interrupted code, perform these steps:

- push all registers you use in your handler on the stack
- process the interrupt
- pop the original register values and execute an IRETD.

6.6.2 To Process an Interrupt and Then Chain

It's often desirable to chain to the original handler for the interrupt (the address of which is obtained with system call 2502h before installing your own handler). For example, a filter on INT 21h might look for one specific function call, and just chain to the original handler for all other functions. This is most simply done as follows:

- execute a PUSHFD to save the current flags
- decrement the stack pointer by 8 bytes to save room for the address of the original handler
- push registers you use
- do any necessary processing
- put the address of the original handler in the reserved slot in the stack
- pop saved register values
- execute an IRETD to transfer control to the original handler with all register values (including EFLAGS) unchanged, and the correct stack frame setup.

6.6.3 To Change Registers When You Return

The simple rules for this depend on which registers you wish to change, as follows:

☛ General registers (EAX-EDX, EBP, ESI, EDI) — simply put what you want in them before executing your IRETD.

☛ CS:EIP, SS:ESP, EFLAGS — Change these on the stack before executing your IRETD. 386|DOS-Extender correctly takes care of the IF bit in EFLAGS on return (please see section 6.6.5).

☛ ES, DS, FS, GS — If the LOADSEG bit in DOSXFLAGS was 0, you can just modify them directly before IRETD, like the general registers. If the LOADSEG bit was set, you must change the values on the stack. If you prefer, you can **always** change the values on the stack. You must then set the LOADSEG bit in DOSXFLAGS before executing the IRETD, to tell 386|DOS-Extender to reload segment registers from the stack frame.

Note: You should **never** change registers in a hardware interrupt handler.

6.6.4 To Chain to the Previous Handler and Regain Control

You can do this normally, by forging an interrupt return address. However, you must provide the full 386 | DOS-Extender stack frame, because the handler you chain to may expect to see it. The easiest way is as follows:

1. pop 386|DOS-Extender umbrella CS:EIP and EFLAGS off the stack, saving them
2. push the CS:EIP and EFLAGS that contains the address where you want to regain control
3. chain to the previous handler for the interrupt
4. when you get control back, push the umbrella CS:EIP and EFLAGS onto the stack again before doing an IRETD.

6.6.5 To Retain Control, Never Returning from an Interrupt

If you never return from an interrupt (e.g., instead of returning to the interrupted code you just jump to a recovery routine in your program), you must make sure any internal resources allocated by 386 | DOS-Extender to handle the interrupt are released. There are two kinds of resources allocated by 386 | DOS-Extender:

- a buffer is allocated for the stack used when the interrupt handler is invoked
- if the interrupt processing includes a mode switch (e.g., if the interrupt originates in real mode, but your handler runs in protected mode), 386|DOS-Extender allocates data structures to hold real mode and protected mode register values.

If you don't arrange for these resources to be freed, eventually 386 | DOS-Extender may run out of buffers and be forced to terminate your program.

If your handler is invoked directly in protected mode without a mode switch, you need to free the buffer used for the interrupt stack. To do this, modify the original CS:EIP and SS:ESP on the interrupt stack frame to point to an entry point in your handler and a stack in memory owned by your application. Then execute an IRETD to allow 386 | DOS-Extender to free the stack buffer and give you back control immediately.

If your handler is invoked with a mode switch (the interrupt originates in real mode), use system call 2501h to free up all internal 386 | DOS-Extender resources. This call frees up all allocated stack buffers and all allocated register data structures. You must switch to a stack in memory owned by your program before making this system call.

Note: The 2501h system call frees **all** allocated data structures, not just the ones used for the current interrupt. Normally, the only allocated structures are for the current interrupt. However, if your program initiates its own mode switches (e.g., by making a cross-mode procedure call to real mode code), keep in mind that the 2501h call will remove the switching context for **all** active mode switches.

6.7 Interrupt Flag Control under DPMI

The instructions that affect the hardware interrupt enable flag (IF) in the EFLAGS register have important interactions with the Input/Output Privilege Level (IOPL). Successful interrupt handling depends on your understanding exactly what occurs under all circumstances.

When the IOPL is less than the current privilege level, input/output causes exceptions. In this circumstance, the instructions that affect IF behave as follows in protected mode:

- PUSHF and PUSHFD always push the hardware-level IF bit
- POPF, POPFD, and IRETD do **not** modify IF, but they also do **not** generate an exception upon attempts to modify IF.

The consequences of these facts vary depending on the environment:

- Under DOS (and VCPI), no problem arises because 386|DOS-Extender always sets IOPL equal to the application's privilege level, so POPF and IRETD can modify IF directly.
- Under DPMI, the host cannot virtualize IF because it doesn't get an exception when the application tries to change IF. This inability is undesirable, because you (the application) **want** the host to be able to virtualize IF, so that if you turn IF off, you succeed in preventing any hardware interrupt handlers you've installed from being invoked.

Therefore, to write an application to be DPMI-compatible, you need to follow these rules:

- When returning from an interrupt handler, it is safe to change IF in the original EFLAGS on the stack and do an IRETD. The 386|DOS-Extender umbrella handler (please see section 6.2) makes the right thing happen. When you get control in an interrupt handler, you can **always** assume IF is off.
- At all other times, use the Get Interrupt Flag system call (function 2534h) to get the current IF setting. Use the CLI/STI instructions (which **can** be virtualized) to modify IF.
- Do **not** assume POPFD or IRETD will change IF, because they won't.
- Do **not** use PUSHFD to obtain the IF setting, because you will get the hardware-level IF instead of the virtual IF that applies to your program.

6.8 PC AT Compatibility Workarounds

There are a number of conflicts that arise during interrupt processing due to the original design of the IBM PC and BIOS. 386 | DOS-Extender provides workarounds for all these conflicts, which include the following:

- hardware interrupts IRQ0-7 and processor exceptions 08h - 0Fh
- hardware IRQ2 interrupt
- BOUND exception and the BIOS print screen system call
- 80387 floating point coprocessor interrupts
- non-maskable interrupts.

This section explains how these conflicts are dealt with.

6.8.1 Hardware Interrupts IRQ0-7 and Processor Exceptions 08h - 0Fh

Interrupt vectors 08h-0Fh are used for hardware interrupts IRQ0-IRQ7. Unfortunately, the 386/486 processor can also generate processor exceptions on these interrupts. 386 | DOS-Extender can always determine whether the source of the interrupt was external hardware or a processor exception.

When you install an interrupt handler in the 08h-0Fh range, you need only specify whether your handler is for the hardware interrupt or the processor exception. You do this by making the appropriate system call to install the handler (please see section 6.3).

6.8.2 Hardware IRQ2 Interrupt

On the original PC and the PC/XT, there was only one interrupt controller with interrupts IRQ0 - IRQ7 supported. The PC AT design removed IRQ2 by dedicating it to the slave interrupt controller. The expansion bus pin connected to IRQ2 on the original PC is connected to IRQ9 on the PC AT, and on PS/2 and EISA machines.

To solve the compatibility problems introduced by this change, the interrupt handler set up for IRQ9 in the BIOS simulates an IRQ2 interrupt; it executes an INT 0Ah instruction to redirect the interrupt to what it presumes to be IRQ2. Programs written for the original PC that use interrupt IRQ2 can thus execute unchanged on the PC AT.

In real mode, there is no problem, because the processor exception for interrupt 0Ah (invalid TSS) can never occur in real mode.

In protected mode, there is no comparable redirection, so a simulated IRQ2 will never occur. An IRQ9 interrupt in protected mode causes 386 | DOS-Extender to switch to real mode and reissue it. The BIOS handler then gets control and issues a simulated IRQ2, again in real mode.

Thus the simulated IRQ2 occurs only in real mode, not in protected mode. Protected-mode handlers that wish to take over this hardware interrupt should therefore always take over IRQ9, never IRQ2.

6.8.3 BOUND Exception and BIOS Print Screen System Call

The 80386 array bounds exception is vectored through interrupt 05h. The BIOS print screen function is also called through software interrupt 05h. This presents problems for programs that use the array bounds checking capability of the 80386.

These include programming languages such as Pascal. Some compilers use the 80386 BOUND instruction, but others, such as MetaWare Professional Pascal, do not. Instead, they do their own array checking, issuing a software INT 05h if array bounds are exceeded.

386 | DOS-Extender resolves this conflict by relocating the BIOS print screen interrupt vector to interrupt 80h. Then, either a true bound exception or a software INT 05h issued in protected mode causes the array bound exception handler to gain control.

Protected-mode programs that wish to invoke the BIOS print screen function must issue INT 80h instead of INT 05h. As an alternative, you can use the -PRIVEC command line switch to select a different interrupt vector relocation for the print screen interrupt, or to disable relocation of the print screen function. A 386 | DOS-Extender system call (250Ch) is available to obtain the print screen interrupt vector selection at run time.

In real mode, 386 | DOS-Extender interprets a software INT 05h as a BIOS print screen request, and therefore invokes the handler to which interrupt vector 80h points. Real-mode programs can thus cause a print screen with either INT 80h or INT 05h. Only a true array bounds exception in real mode will cause the bound exception handler to gain control.

6.8.4 Coprocessor Interrupts

The 80386 coprocessor exception, intended for use with the Intel 80287 and Intel 80387 floating point coprocessors, is vectored through interrupt 10h. This is also the software interrupt used to invoke the BIOS video functions.

However, this conflict turns out not to be a problem, because of the way an 80287 or 80387 on a PC AT, PS/2, or EISA machine is connected. Instead of going through interrupt 10h, coprocessor exceptions are vectored through hardware interrupt IRQ13.

Therefore, interrupts vectored through 10h always invoke the BIOS video functions, and programs that will handle coprocessor exceptions must take over IRQ13 rather than interrupt 10h.

6.8.5 Non-Maskable Interrupt

The non-maskable interrupt (NMI) is a dedicated input to the 80386, always vectored through interrupt 02h. The NMI is generally not used on PCs, but you can add hardware connecting this interrupt to a pushbutton switch. Pressing the switch then gives control to the NMI handler at any time, for debugging or other purposes.

Programmers using the NMI must understand a compatibility problem with the 80287 or 80387 coprocessor exception, which is vectored through hardware interrupt IRQ13. For backward compatibility with the original PC, the BIOS handler for IRQ13 performs an end of interrupt (EOI) to the interrupt controller and then issues a software INT 02h.

There are at least two workarounds that enable using both the NMI and the numeric coprocessor. The first is simply to mask all exceptions on the 80287 or 80387, if present, so that an IRQ13 will never be generated. The second is to take over interrupt IRQ13 directly, so that the BIOS handler for IRQ13 never gains control.

Chapter 7

Mixing Real- and Protected-Mode Code

7.1 Introduction

The three problems that must be solved when mixing real- and protected-mode code in a single program are as follows:

- getting both the real-mode and the protected-mode code into memory simultaneously, while ensuring that the real-mode code ends up in conventional memory
- passing data back and forth between real and protected mode
- making intermode procedure calls.

Each of these problems has more than one viable solution. This chapter discusses the available options and offers some criteria for making a choice.

7.2 Program Loading

There are three options available for loading real-mode code and protected-mode code that need to communicate with each other, as follows:

- Link the real-mode code and the protected-mode code into a single task image. This is the method used in the TAIL.ASM sample program that uses the Microsoft mouse driver (please see section 7.9.2). Conceptually, this method seems the simplest. However, you must work with segment ordering in the link and memory allocation at load time to make sure the code gets loaded correctly. Section 7.2.1 discusses these issues. Since this method requires the use of the -REALBREAK switch, it is not compatible with most DPMI environments.

Also, this approach works well only if the real-mode code is written in assembly language. Combining the runtime libraries for real-mode and protected-mode compilers in the same link would usually result in many multiply defined symbols.

However, most protected-mode programs needing some real-mode code only require a small amount, to communicate with an existing real-mode hardware driver or the like. Meeting such requirements using assembly language is appropriate, and this method is probably the best to use.

☛ During execution of the protected-mode program, have it make an EXEC system call to load the separate real-mode program into memory. This is the method used by the GDEMO.C sample program that uses the Microsoft C 5.1 graphics library in real mode (please see section 7.9.1). With protected-mode and real-mode code in separate .EXE files, there is no problem in using different compilers or high-level languages, if desired.

The difficulty is communicating between the two programs; not being linked together, neither one has the addresses for procedures and data in the other. Sections 7.2.4 and 7.3 discuss issues of interprogram communication.

The only configuration issue with this approach is leaving enough conventional memory free to load the real-mode program. Section 7.2.2 discusses the appropriate switches and function calls.

☛ During execution of the real-mode program, have it perform an EXEC to the protected-mode program. With this method, there is no problem with conventional memory allocation, since the real-mode program gets loaded first.

The same communication difficulties exist with this option as with the second option. Sections 7.2.4 and 7.3 discuss issues of interprogram communication. When debugging, there is also the annoyance of changing the EXEC system call to EXEC to the debugger rather than to 386|DOS-Extender.

7.2.1 Linking Real- and Protected-Mode Code Together

Linking real-mode and protected-mode code in the same program is an appropriate choice when the amount of real-mode code to be written is relatively small and can be written in assembly language. The real-mode code is placed in USE16 segments and then assembled with the 80386 as the target. This creates code that will execute correctly in real mode. There is no restriction on addressing modes, operand sizes, or available instructions, as real mode on the 80386 allows the same addressing modes and operand sizes as protected mode, and the assembler will automatically generate any necessary override bytes, etc. The program is then linked with the 80386 as the target machine, as usual for protected-mode programs.

There are, however, some restrictions that must be observed. When 386 | LINK creates a protected-mode program, it combines all the code, data, and stack segments in the program into a single large program segment. Because segmented links are not supported, there can be no segment fixups required by the program. This means that all the real-mode code must reside in a single segment, and only NEAR calls and jumps can be performed by the real-mode code. This restriction effectively limits the size of the real-mode code to 64 KB (the maximum size of a segment in real mode).

The next problem to be solved is making sure that all the real-mode code and data gets loaded in conventional memory. The real-mode code must be loaded in conventional memory in order to execute in real mode. In addition, there is typically some subset of the program's data that is used for communication between the real-mode and protected-mode portions of the program. This data must be loaded in conventional memory, so that it is possible for the real-mode code to access it.

All of the code and data to be loaded in conventional memory must be positioned at the beginning of the .EXP file, when the program is linked. This is done by using the segment ordering rules applied by 386 | LINK (please see the *386 | LINK Reference Manual*). The real-mode code, and the data that needs to be accessed in real mode, should be placed in their own segments. The order of the segments in the first object file processed by the linker, along with the segment classes and other linker ordering rules, will then determine how the segments will be ordered in the .EXP file. The objective is to end up with an .EXP file that is ordered as shown in Figure 7-1:

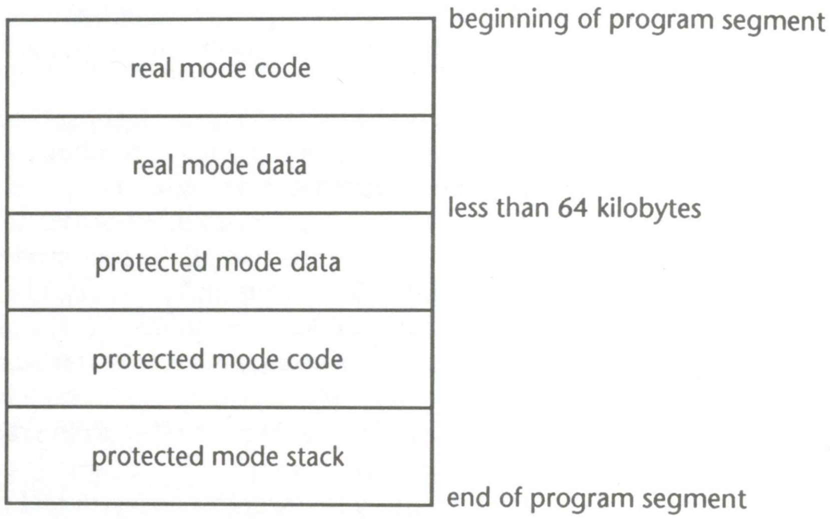


FIGURE 7-1
SEGMENT ORDERING IN AN .EXP FILE

Since the linker combines everything into one big segment in the link, all offsets are from the beginning of this large program segment. When executing in real mode, the CS and DS segments are set up with the real-mode segment paragraph address of the beginning of the program segment, and normal references to locations in the real-mode code and data work correctly. The only requirement is that the real-mode code plus data be less than 64 KB in size.

When a real-mode procedure gains control, the CS register is loaded with the real-mode paragraph address of the beginning of the program segment. The real-mode procedure need only copy this address from CS to DS, and then it can directly access all of the real-mode data in the program.

When executing in protected mode, the segment descriptors accessed by the segment selector values in CS and DS map the entire program segment, as usual. It is, therefore, possible to directly reference data in both the real-mode and protected-mode data segments.

The use of the assembler ASSUME directive to make data references work correctly is somewhat nonintuitive, due to the fact that the program source code is segmented; but when it is linked, everything is combined into a single segment. Because the linker relocates all addresses relative to the beginning of the single program segment, all data references work correctly, in both real mode and protected mode, when the DS segment register is set up as described above. The only problem is communicating this fact to the assembler, so that it will not complain!

One solution is to use the ASSUME directive to tell the assembler that DS points to whichever data segment you currently want to access. A better solution, however, is to place both the real-mode data segment and the protected-mode data segment in a group, and then use the ASSUME directive to say the DS register points to the group. This will allow the assembler to correctly assemble references to data in both real and protected mode without having to sprinkle ASSUME directives throughout the program. This is the reason the protected-mode data segment is shown next to the real-mode data segment instead of following the protected-mode code segment.

Once the segment ordering has been set up correctly, the only remaining problem is to make sure that all the real-mode code and data is actually loaded in conventional memory. This is done either when the program is linked, or when it is run under 386 | DOS-Extender, by using the -REALBREAK command line switch. This command line switch specifies how much of the .EXP file **must** be loaded into contiguous conventional memory. It is generally easiest to use this switch when the program is linked, since it can then be done symbolically by giving the name of a public symbol at the end of the real-mode data segment. If this switch is used when the program is run under 386 | DOS-Extender, an absolute number must be given, which means that you must use the map file generated by the linker to calculate how large the real-mode code and data are. Using -REALBREAK is incompatible with most DPMI environments.

The TAIL.ASM program listing in section 7.9.2 gives an example of how all this works in practice, including segment ordering and segment register initialization.

7.2.2 EXEC a Real-Mode Program from a Protected-Mode Program

The second alternative is to have the protected-mode program load the real-mode program into memory by making an EXEC system call (4Bh or

25C3h). Once the real-mode program gets control, it can make calls to the protected-mode program, which, of course, is still in memory. If this method is used, there are no restrictions on the size of the real-mode program or the number of segments it has. There are also no unusual segment ordering problems with this approach. The only requirement is that the protected-mode program be linked or run with the -MINREAL and/or -MAXREAL switches to make sure that enough conventional memory is available to load the real-mode program. Alternatively, you can use system calls 2525h or 2536h to free up conventional memory when needed.

Although the setup and program loading problems are much simpler with this approach, the problem of communication is more difficult, because the two programs do not have the addresses of each other's procedures and data. In addition, the real-mode program cannot directly access any of the protected-mode program's data, because it may not be in conventional memory. The solutions to these problems are addressed in sections 7.3 and 7.4.

The sample program GDEMO.C, included on the product distribution disks and described in section 7.9.1, uses this program loading technique.

7.2.3 EXEC a Protected-Mode Program from a Real-Mode Program

The third approach to program loading is to start up a standard real-mode program that will perform an EXEC to a protected-mode program by calling the MS-DOS system function 4Bh. Once the protected-mode program gains control, it can make calls to the real-mode program, which, of course, is still in memory. Nothing special need be done in the program loading phase with this approach; there are no memory allocation or program placement problems. As with the previous solution, however, the communications problems are more difficult. Please see the following sections for details.

7.2.4 Maintenance of Two PSPs

One complication of linking a mixed-mode program as separate real-mode and protected-mode programs is that each program has its own program segment prefix (PSP). The PSP is used by MS-DOS to maintain information about the program, and is described in section 3.3 and References 3 and 11 in the Preface.

MS-DOS has the concept of the "current" PSP. When a program is loaded into memory to be executed, its PSP is set up as the current PSP. If it then EXECs to a second program, the child program's PSP is set up as the current PSP until the child terminates. Then the parent's PSP again becomes the current PSP and control is transferred back to the parent.

A real-mode and protected-mode program, loaded separately, can call back and forth to each other as described in section 7.4. The current PSP is always the one for whichever program is loaded into memory second (with an EXEC call). MS-DOS uses this PSP for the entire time both programs are in memory, regardless of which is actually executing. Usually this poses no problems. However, there are circumstances under which this behavior has undesirable side effects.

One of the functions of the PSP is to maintain a list of disk files that have been opened by the program. When the program terminates, all of the open files are automatically closed. To see how this could create a problem, consider the following example. Suppose a protected-mode program is run initially, and it then EXECs to a real-mode program with which it communicates. If files are opened (by either program) while the real-mode program is executing, they will be referenced in the real-mode program's PSP, and not in the protected-mode program's PSP. When the real-mode program terminates, all of those files are automatically closed and the protected-mode program can no longer access them. This may or may not be desirable, depending on whether the protected-mode program needs to access those files again.

If the above example or some other use of the PSP causes problems, each program must take responsibility for switching to its own PSP each time it gains control. This is done with the Get PSP and Set PSP system calls. The Set PSP function (50h) is exactly the same in real mode and protected mode; it simply passes MS-DOS the real-mode paragraph address of the PSP to set up as the current PSP. Use function 51h to get the real-mode segment address of the current PSP. An example of programs switching PSPs is given in the GDEMO.C sample program (please see section 7.9.1).

Note that when a program calls MS-DOS to terminate, its PSP must be set up as the current PSP. One of the fields in the PSP is a pointer to the program's parent. This is how MS-DOS knows where to return control when a program terminates.

7.3 Passing Data Between Modes

386 | DOS-Extender provides three methods for passing data on intermode procedure calls. The first is to simply pass data on the stack. Intermode calls from real to protected mode retain the same stack, so any data placed there by the real-mode code is automatically accessible by the protected-mode procedure. Intermode calls from protected to real mode supply a parameter to 386 | DOS-Extender specifying how much data to copy. That many bytes are then copied from the protected-mode procedure's stack onto the stack given to the real-mode procedure. The stack setup and usage on intermode procedure calls is described in more detail in section 7.4.

The second method is to use the application data buffer. During initialization, 386 | DOS-Extender optionally allocates a buffer that is guaranteed to reside in conventional memory. The size of this buffer is controlled by the -CALLBUFS command line switch (please see section 2.6) and can range from 0 to 64 KB in size. This buffer is for the exclusive use of the application program in any manner desired; 386 | DOS-Extender will never use this buffer. A system call is provided to get both the real-mode and protected-mode address of this buffer at run time. Intermode procedure calls can, therefore, define any protocol they desire for passing data back and forth in this buffer.

A third option for passing data between modes is to use the 80386 registers. 386 | DOS-Extender does not modify any registers on intermode procedure calls, except as noted in section 7.4.

7.4 Intermode Control Transfer

There are two techniques available for transferring control between real mode and protected mode:

- use the 386|DOS-Extender intermode procedure call facilities

- ☛ use software interrupts to switch between modes.

It's best to use procedure calls, because using software interrupts requires choosing interrupt vectors to use for mode switching. Taking over arbitrary interrupts for your own use means your program will be incompatible with other products that happen to use the same software interrupts for a different purpose.

With either control transfer method, keep in mind that cross-mode calls cannot be nested to arbitrary depths because 386 | DOS-Extender allocates internal data structures for each mode switch, and only frees them up when the event that caused the mode switch (interrupt or cross-mode procedure call) returns.

The data structures most likely to be exhausted are the buffers used for stacks when an interrupt occurs in protected mode. One of these buffers is allocated for each switch from protected mode to real mode (even a cross-mode procedure call is initiated with a software INT 21h to make the 386 | DOS-Extender system call). A stack buffer is also allocated when an interrupt taken over with system call 2506h causes a switch from real mode to protected mode. A cross-mode procedure call from real mode to protected mode does **not** allocate a stack buffer, since the real mode stack is used for the protected mode routine (please see section 7.4.1).

The size and number of the internal stack buffers are controlled by the -NISTACK and -ISTKSIZE command line switches. The default is six buffers of 1 KB each. You should always leave three stack buffers available for the worst-case situation: a 386 | DOS-Extender system call (INT 21h function 25xxh) is made. As part of its processing, it makes an INT 21h DOS call (which is done by several of the 386 | DOS-Extender system calls). While the MS-DOS call is executing, a hardware interrupt occurs. Three stack buffers are used: one each for the two software INT 21hs, and one for the hardware interrupt.

The default stack buffers count of six therefore leaves three available to you for nested mode switches. If you need to nest mode switches more than three levels deep (one level deep if you use software interrupts to switch from real to protected mode), use the -NISTACK switch to allocate extra stack buffers.

7.4.1 Intermode Procedure Calls

The 386 | DOS-Extender system functions 250Eh and 2510h call a real-mode function from protected mode. They differ in that function 2510h permits the caller to specify the real-mode values of all registers, including the segment registers. Function 250Eh sets the DS register to the same value as CS and leaves all other segment registers undefined.

These functions take as inputs the FAR (segment:offset) real-mode address of the procedure to call, and a count. The count represents the number of 2-byte WORDs to copy from the protected-mode stack to the stack given to the real-mode procedure.

The following are true when the real-mode procedure gains control:

- all general registers are unchanged
- CS:IP is, of course, equal to the procedure entry point
- SS:SP points to an internal 386|DOS-Extender stack buffer
- for function 2510h, the caller specifies the real-mode DS, ES, FS, and GS segment register values
- for function 250Eh, DS is initialized to the same value as CS, because this is desirable for programs in which the real- and protected-mode code have been linked together.

The real-mode stack size is controlled by the -ISTKSIZE command line switch and defaults to 1 KB. The stack will have whatever data were specified to be copied from the protected-mode stack, followed by a FAR real-mode return address. For example, if three 2-byte WORDs were to be copied, the stack would look as follows:

	Offset	Size	Description
	08h	WORD	third 2-byte parameter copied
	06h	WORD	second 2-byte parameter copied
	04h	WORD	first 2-byte parameter copied
	02h	WORD	real-mode CS return value
SS:SP—>	00h	WORD	real-mode IP return value

When the real-mode procedure completes its processing, it returns control to the protected-mode procedure by executing a FAR return, with the following results:

- all general registers are as they were left by the real-mode procedure
- all segment registers are restored to their original protected-mode values
- the processor flags are preserved.

Processor flags are preserved in both directions on the call. However, on return to protected mode, the carry flag indicates the success or failure of the system call. Please see the description of system calls 250Eh and 2510h in Appendix B for a complete description of the environment of a call through to real mode.

For real-mode code to call a protected-mode procedure, it executes a FAR call to a 386 | DOS-Extender real-mode routine. The address of this 386 | DOS-Extender routine is obtained dynamically at run time by calling system function 250Dh from protected mode. Before making the FAR call, the real-mode procedure pushes any parameters to be passed to the protected-mode procedure, a real-mode FAR pointer to a data structure containing the protected-mode segment register values, and the address of the protected-mode procedure. (Please see system call 250Dh for complete documentation.) The stack looks as follows when making the FAR call to the 386 | DOS-Extender routine:

	Offset	Size	Description
	0Ah		arguments to protected-mode routine
	06h	DWORD	real-mode FAR pointer to data structure with protected-mode register values
	04h	WORD	protected-mode routine segment selector
SS:SP—>	00h	DWORD	protected-mode routine segment offset

When this FAR call is made, 386 | DOS-Extender pops the real-mode return address, the data structure pointer, and the protected-mode procedure address off the stack and saves them. It then switches to protected mode and transfers control to the protected-mode procedure, with the following results:

- ☛ All the general registers are unchanged.
- ☛ SS:ESP points to the real-mode stack. (This is done by setting SS to a 386|DOS-Extender segment that maps the first MB of MS-DOS memory, and setting ESP to the linear address of the stack). Parameters may, therefore, be directly passed to the protected-mode procedure on the stack.
- ☛ DS, ES, FS, and GS are loaded with the values specified in the data structure.

When the protected-mode procedure completes its processing, it terminates by executing a FAR return. 386 | DOS-Extender then returns control to the real-mode procedure. All general registers are as they were left by the protected-mode procedure, and all segment registers are restored to their original real-mode values. The processor flags are preserved in both directions on the call. Please see the description of system call 250Dh in Appendix B for a complete description of the environment on a call through to protected mode.

7.4.2 Switching Modes with Interrupts

Programs that will switch modes with software interrupts must have some initialization code that takes over interrupt vectors to be used for mode switching. At a minimum, two interrupt vectors are required. One interrupt is used to switch from protected mode to real mode and is referred to in this section as the "real interrupt." The other interrupt is used to switch from real to protected mode and is referred to as the "protected interrupt."

The initialization routine must execute in protected mode. It takes over the protected interrupt with system function 2506h, which sets up a protected-mode handler to get control whenever the interrupt occurs. The real interrupt is taken over with system function 2505h, which sets up a real-mode handler to get control when the interrupt occurs.

Alternatively, a real-mode initialization routine can call MS-DOS function 25h to take over the real interrupt. Real-mode code cannot take over the protected interrupt.

Once the initialization has been performed, the protected-mode code can transfer control to the real-mode interrupt handler at any time by issuing the real interrupt as a software interrupt. When the real-mode interrupt handler gets control, all the general registers and processor flags have the same values they had when the interrupt was issued in protected mode, except that the interrupt flag is off (interrupts disabled). The SS:SP registers are set up to point to a stack memory buffer maintained internally by 386 | DOS-Extender; the size of this buffer is controlled by the -ISTKSIZE command line switch and defaults to 1 KB. The DS, ES, FS, and GS registers are undefined.

When the real-mode handler completes its processing, it exits with the IRET instruction. 386 | DOS-Extender then returns control to the protected-mode procedure, with all general registers and processor flags (except the interrupt and trap flags) set as they were left by the real-mode handler. The original protected-mode values of all the segment registers are restored.

Real-mode code can transfer control to the protected-mode interrupt handler at any time by issuing the protected interrupt as a software interrupt. When the protected-mode handler gets control, all the general registers and processor flags have the same values they had when the interrupt was issued in real mode, except that the interrupt flag will be off (interrupts disabled). The SS:ESP registers are set up to point to a stack memory buffer maintained internally by 386 | DOS-Extender.

If desired, the protected mode handler can look at parameters on the real mode stack by obtaining the real mode stack address from the interrupt stack frame. Please see section 6.5 for details.

When the protected-mode handler has completed its processing, it exits by executing the IRETD instruction. 386 | DOS-Extender then returns control to the real-mode procedure, with all general registers and processor flags (except the interrupt and trap flags) set as they were left by the protected-mode handler. The original real-mode values of all the segment registers are restored .

7.5 Typical Program Organization

When designing a program that will mix real- and protected-mode code, it is possible to mix and match the techniques described above. This section describes some of the more straightforward and useful ways to organize mixed-mode programs.

7.5.1 Linking Real- and Protected-Mode Code Together

Programs that mix real- and protected-mode code in a single .EXP file typically have an initialization routine that does the following:

- obtains the MS-DOS address of the beginning of the program segment with system call 250Fh
- calls system function 250Dh to obtain the real-mode address of the 386|DOS-Extender routine used to switch from real mode to protected mode
- saves this information in global data that the real-mode code can access.

The program then has all the information it needs to switch modes with the 386 | DOS-Extender intermode procedure calls. Data transfer is done either on the stack or through the global data that can be accessed by the real-mode code.

This method is illustrated in the TAIL.ASM example program (please see section 7.9.2).

7.5.2 Starting with Protected-Mode Code

Some programs are set up to have the protected-mode program load the real-mode program into memory with a 4Bh EXEC system call. They typically have an initialization routine that performs the following steps:

1. The protected-mode program calls system function 250Dh to obtain the real-mode address of the 386|DOS-Extender routine used to switch from real mode to protected mode, and to obtain the real- and protected-mode addresses of the data buffer used for passing data on intermode calls.
2. It then calls system function 4Bh to EXEC to the real-mode program, using the command line to pass (a) the real-mode address of the 386|DOS-Extender routine used to call through to protected mode, (b) the real-mode address of the intermode data buffer, and (c) the protected-mode address of the interface routine in the protected-mode program.
3. The real-mode program then gets control and performs its own initialization, and then calls the protected-mode interface routine to give control back to the protected-mode program, passing it the real-mode address of the interface routine in the real-mode program.

The two programs can now continue executing, calling back and forth to each other and passing data either on the stack or in the intermode data buffer. When the program has completed its task, the protected-mode program returns from the call to the interface routine made in step 3. The real-mode program then terminates by calling MS-DOS function 4Ch, and the protected-mode routine gets control again at the instruction following the 4Bh EXEC system call.

This method is illustrated in the GDEMO.C example program (please see section 7.9.1).

7.5.3 Starting with Real-Mode Code

Some programs are organized to have a real-mode program start first and then perform an EXEC to a protected-mode program. They typically execute the following initialization procedure:

1. One or more interrupts are taken over by calling the standard real-mode MS-DOS get and set interrupt vector system functions. These interrupts are used by the protected-mode program to transfer control back to the real-mode program.
2. The standard MS-DOS system EXEC function 4Bh is called to load the protected-mode program and start it executing.
3. The protected-mode program takes over one or more interrupts with system function 2506h. These interrupts are used by the real-mode program to transfer control to the protected-mode program.

Once this initialization is complete, the real- and protected-mode code just use the prearranged software interrupts to switch modes. Data are passed on the stack or in registers. Alternatively, the protected-mode code can call system function 250Dh to obtain the address of the intermode call data buffer, and then pass the address to the real-mode code, so that the data buffer can be used for passing data when switching modes with a software interrupt.

7.6 Making Arbitrary Real-Mode System Calls

386 | DOS-Extender extends all of the documented MS-DOS system calls, and most of the BIOS system calls, so that they can be made directly from protected mode. However, some programs also need access to undocumented MS-DOS functions with pointers, or the ability to make system calls to another program, such as a network driver installed in memory at system boot time.

Any system calls that use only the general registers (no segment registers) for inputs and outputs are called from protected mode in the same way they are called from real mode (please see section 6.1). If the system call uses segment registers, then additional processing is required by the protected-mode program.

386 | DOS-Extender system call 2511h issues a real-mode interrupt with all registers, including the segment registers, specified. It also returns the values left in the segment registers by the real-mode interrupt handler. This system call makes it possible to issue any interrupt that can be issued by a real-mode program.

The other half of the problem is buffering data. System calls that use segment registers usually do so because they require a pointer to a buffer containing data. This data buffer must be in conventional memory, so that real-mode code can address it. A data buffer guaranteed to be in conventional memory can be allocated by forcing part of the protected-mode program to be loaded in conventional memory (please see section 7.2.1), or by calling MS-DOS directly to allocate memory (please see section 7.7). Probably the simplest method, however, is to use the -CALLBUFS command line switch to allocate a data buffer in conventional memory at initialization time, and to call system function 250Dh at run time to obtain the real- and protected-mode addresses of the data buffer.

When a system call returns a pointer to a data buffer, the data can be accessed using segment selector 0034h, which maps the first MB of conventional memory.

7.7 Allocating Conventional Memory

The standard MS-DOS memory allocation calls (48h, 49h, and 4Ah) are handled entirely by 386 | DOS-Extender. These calls allocate and free blocks of memory, which may be either in extended memory or in conventional memory. Memory blocks allocated by 386 | DOS-Extender consist of 4-KB memory pages, which may or may not be contiguous in physical memory (please see Chapter 5).

Some programs may require memory blocks which are guaranteed to be in conventional memory and/or to be physically contiguous. 386 | DOS-Extender provides a set of system calls (2537h, 25C0h, 25C1h, and 25C2h) to call the MS-DOS memory allocator. Memory allocated with these calls can be addressed in protected mode by using segment selector 0034h, which maps the first MB of conventional memory.

Note that the -MINREAL and/or -MAXREAL command line switches can be used to force 386 | DOS-Extender to leave some conventional memory free at initialization time. Alternatively, you can use system calls 2525h or 2536h to free up conventional memory. If this is not done, any attempts to allocate conventional memory will fail, returning an out-of-memory error.

7.8 Protected-Mode Memory Resident Programs

Memory resident programs (also referred to as terminate and stay resident, or TSR, programs) are programs that take over one or more hardware interrupts and then exit to MS-DOS, using a system call that leaves the code and data for the program in memory. Then, other programs can be run from MS-DOS, and the TSR program can gain control whenever a certain condition occurs. Most TSR programs take over the keyboard interrupt and pass most keystrokes along to the previous keyboard handler, only taking control when a specific keystroke designated as the "hot key" for that program is pressed.

There are two system calls that can be used to terminate and stay resident: INT 21h function 31h, or INT 27h. Before making one of these calls, the TSR program should free up any memory it does not require for its own use, so as much memory as possible will be available for other programs. When a protected-mode program makes the TSR system call, both the protected-mode program and 386 | DOS-Extender stay resident in memory. Protected-mode TSR programs must, therefore, take the 386 | DOS-Extender memory requirements (please see Chapter 5), as well as its own, into account.

Protected-mode TSR programs are usually linked with the -MAXDATA 0 and -MAXREAL FFFFh command line switches. The -MAXDATA 0 switch means that no additional memory is allocated to the program beyond that required to load the program into memory (by default, the program gets all the memory on the machine). The -MAXREAL FFFFh switch means that as much conventional memory as possible is left free to be used by other programs. If this switch is not used, 386 | DOS-Extender allocates all conventional memory to be used as part of the memory pool available to the protected-mode program (please see section 5.2.1).

7.9 Example Programs

The product distribution disks include several sample programs (please see Appendix D) that demonstrate the techniques outlined in this chapter. The sections below describe some of these sample programs.

7.9.1 GDEMO.C Graphics Example

The GDEMO.C program demonstrates making calls from a protected-mode C program to a real-mode graphics subroutine library. The library used is the graphics library distributed with the Microsoft C 5.1 compiler. To build this example, you need both the MetaWare High C-386 compiler and the Microsoft C 5.1 compiler.

This program starts up in protected mode, and then uses the EXEC call to load a real-mode program that includes the Microsoft graphics library routines. It then sets up a mixed-mode interface that allows the protected-mode C code to make C subroutine calls to the real-mode graphics library.

The sample program can be built in two different ways, using two different techniques for passing data. (For details on how to build the program, see the README file included on the distribution disks.)

One method demonstrated is passing data in a data buffer linked into the protected-mode program. When the protected-mode program is linked, the -REALBREAK switch is used to ensure that the data buffer is loaded in conventional memory where the real-mode program can access it. The advantage of this approach is that the protected-mode program can use NEAR pointers (which is the normal case for 32-bit protected-mode programs) to access the data buffer. The disadvantage is that -REALBREAK is not supported in most DPMI environments, including Windows 3.0, so programs built in this way cannot run under Windows 3.0 enhanced mode.

The second method uses the -CALLBUFS switch to allocate a conventional memory buffer when the protected-mode program is loaded. The protected-mode program then constructs a FAR pointer, using segment 0034h, to access the data buffer. The advantage of this approach is DPMI compatibility. The disadvantages are the extra overhead of using FAR pointers, and the fact that not all 386 compilers include full support for FAR pointers.

7.9.2 TAIL.ASM Microsoft Mouse Example

This section gives an example of a fairly simple program that mixes real-mode and protected-mode code. The program is used with the Microsoft serial mouse. All it does is move a cursor around on the screen as the mouse is moved, printing a character at the current screen location each time the right mouse button is pressed. Pressing the left button and then the right button toggles between the asterisk (*) and the space ( ) as the output characters. Pressing the left mouse button twice causes the program to terminate. This program is included on the 386 | DOS-Extender SDK distribution disks as the file TAIL.ASM.

This program demonstrates combining protected-mode and real-mode code in a single program. It is organized so that most of the code executes in protected mode. Only the handlers invoked by the mouse driver, and a little bit of initialization code, execute in real mode. The mode switching technique used is intermode procedure calls (please see section 7.4.1). The program is somewhat more complex than necessary, in the interests of maximizing the number of different techniques illustrated.

The entire source code to TAIL.ASM is listed below. Following the program listing is a discussion of the salient features of the program.

```
;
; TAIL.ASM - This is an example program that uses
; the Microsoft mouse driver, and also demonstrates
; several techniques for mixing real and protected
; mode code in the same program.
;
; FUNCTIONALITY - The program is used to draw rough
; pictures on the screen using the mouse.  The
; program is always in one of two modes - character
; output mode, or toggle mode.  When in character
; output mode pressing the right mouse button will
; output a character at the current cursor position.
; Pressing the left mouse button causes the program
; to enter toggle mode.  When in toggle mode,
; pressing the right mouse button will cause the
; character that gets output to be toggled between
; a '*' and a ' ' (space).  Pressing the left mouse
; button in toggle mode causes the program to exit.
; The program starts up in character output mode,
; with the output character set to '*'.  The program
; is used to draw rudimentary pictures with '*'s,
; toggling the output character to a space to
; erase mistakes.
;
; IMPLEMENTATION - The program demonstrates mixing
; real and protected mode code in a single .EXP
; file (in this instance, in a single source code
; file).  The real mode code and data is linked at
; the beginning of the program, and the -REALBREAK
; switch is used at link time to specify how much
; of the program must be loaded in conventional
; MS-DOS memory.  The program has three routines
; that run in real mode - a setup routine that calls
; the mouse driver to specify the initial mouse
; interrupt handler, and two mouse interrupts,
; which correspond to the two program states,
; character input mode and toggle mode.  The
; program demonstrates how to make intermode
; procedure calls passing data both on the stack
; and in registers, and how to issue real mode
; interrupts from protected mode with arbitrary
; real mode segment register values.
;

;
; Segment ordering and attributes.  We make sure the
; real mode code and data comes first.  The real and
; prot mode data are together so they can be grouped.
;
```

```
rmcode   segment byte public use16
rmcode   ends
rmdata   segment dword public use16
rmdata   ends
pmdata   segment dword public use32
pmdata   ends
pmcode   segment byte public use32
pmcode   ends
stack    segment dword stack use32
stack    ends
dgroup   group   rmdata,pmdata

;
; Constants
;
TRUE     equ     1
FALSE    equ     0

;
; Make entry points and data global so they
; can be accessed by the debugger
;
public   main,init,cleanup,rm_setup,ch_hndlr
public   tog_hndlr,print_ch
public   donef,call_prot,vid_mode

;
; Give program a 4K stack
;
stack    segment
         db      4096 dup(?)
stack    ends

;
; Data that needs to be accessed in both real
; mode and protected mode
;
rmdata   segment

donef    dd      ?        ; flags when done
outp_ch db       ?        ; current output char
                          ; ('*' or ' ')
call_prot dd     ?        ; addr of routine to
                          ; call thru to prot mode
code_sel dw      ?        ; prot mode code selector for this pgm

chndlrmsg db     'Output char at cursor error'
         db      0Dh,0Ah,'$'
```

```
;
; Symbol marking end of real mode code & data,
; used at link time to specify the real mode
; code & data size
;
        public  end_real
end_real label  byte

rmdata  ends

;
; Data that is only accessed in prot mode
;
pmdata  segment

rm_seg  dw      ?       ; real mode segment addr
                                ; of start of
                                ; pgm segment
vid_mode db     ?       ; saved video mode

drivmsg db      'Mouse driver not present'
        db      0Dh,0Ah,'$'
b2msg   db      'Mouse must have 2 buttons'
        db      0Dh,0Ah,'$'
realmsg db      'Real mode code not addressable'
        db      0Dh,0Ah,'$'
callmsg db      'Call to real mode failed'
        db      0Dh,0Ah,'$'
setupmsg db     'Real mode setup routine error'
        db      0Dh,0Ah,'$'

pmdata  ends

;
; Because of data grouping, this ASSUME will
; allow the assembler to correctly reference
; data in both real mode and prot mode code
;
        assume  ds:dgroup
page
;
; Protected mode code
;
        assume  cs:pmcode
pmcode  segment
```

```
;*****************************************************
; main - sets up the mouse handler, then just loops,
;       processing mouse button hits until the left
;       one is pressed twice.
;*****************************************************
main    proc    near                ; program entry point

        call    init                ; init mouse
        cmp     eax,TRUE            ; branch if error
        je      short #err          ;

        mov     donef,FALSE         ; init done flag
        mov     outp_ch,'*'         ; init output char
#wait:
        cmp     donef,FALSE         ; loop until handler
        je      #wait               ; sets flag

        call    cleanup             ; clean up mouse
        mov     al,0                ; return success

#exit:
        mov     ah,4Ch              ; exit to DOS
        int     21h                 ;

#err:
        mov     al,1                ; return error
        jmp     #exit               ;

main    endp

;*****************************************************
; init - This routine checks for the presence
;       of the mouse driver, and if it is present,
;       initializes it and sets up the real mode
;       handler to get control when a mouse button
;       is pressed.  It also sets up an appropriate
;       video mode, and gets the address of the
;       real mode DOS-Extender routine which is
;       used to call through to protected mode.
;
; Returns:      TRUE if error
;               FALSE if success
;*****************************************************
init proc near

        push    es                  ; save regs

;
; Get code segment selector for this program
;
        mov     code_sel,cs
```

```
;
; Save the current video mode, and set the video mode
; to 80x25 B&W (8x8 cell size).
;
        mov     ah,0Fh              ; get video state
        int     10h                         ;
        mov     vid_mode,al                 ;
        mov     ax,0002h            ; set video state
        int     10h                         ;


;
; Get address of DOS-Extender routine to use to
; call through from real mode to protected mode
;
        mov     ax,250Dh            ; get real mode link
        int     21h                         ; info
        mov     call_prot,eax       ; save proc address


;
; Check for presence of mouse driver, take error
; exit if not present, initialize it if present
;
        mov     cl,33h              ; get mouse real mode
        mov     ax,2503h                    ; int vector
        int     21h                         ;
        cmp     ebx,0               ; branch if null
        je      #no_driver                  ;
        xor     eax,eax             ; branch if vector
        mov     ax,bx                       ; just points
        and     ebx,0FFFF0000h              ; to an IRET
        shr     ebx,12                      ;
        add     ebx,eax                     ;
        mov     ax,0034h                    ;
        mov     es,ax                       ;
        cmp     byte ptr es:[ebx],0CFh      ;
        je      short #no_driver            ;
        mov     ax,0                ; Initialize mouse
        int     33h                         ; driver
        cmp     ax,0                ; branch if not
        je      short #no_driver            ; installed
        cmp     bx,2                ; pgm requires 2
        jne     short #buttons              ; buttons
```

```
;
; Call through to real mode to set up interrupt
; handler for mouse driver.  This call demonstrates
; passing data both on the stack and in registers,
; and getting back a value in a register.
;
        mov     ax,ds               ; get real mode para
        mov     es,ax                       ; address of
        xor     ebx,ebx                     ; program
        lea     ecx,end_real                ; segment
        mov     ax,250Fh                    ;
        int     21h                         ;
        jc      short #not_real ; branch if error
        test    ecx,0FFFFh      ; branch if not on
        jnz     short #not_real             ; para boundary
        mov     ebx,ecx         ; EBX = real mode addr
        lea     bx,rm_setup                 ; of rm_setup
        shr     ecx,16          ; pass hndlr seg addr
        push    cx                          ; on stack
        mov     rm_seg,cx       ; save seg addr
        lea     dx,ch_hndlr     ; pass offset in DX
        mov     ecx,1           ; 1 WORD on stack
        mov     ax,250Eh        ; call rm_setup
        int     21h                         ;
        jnc     short #ok       ; branch if success
        add     esp,2           ; pop args and take
        jmp     short #bad_call             ; error exit
#ok:
        add     esp,2           ; pop args off stack
        cmp     ax,FALSE        ; branch if error
        jne     short #setup_err            ;

;
; OK, all initialized - display mouse cursor and exit
;
        mov     ax,1            ; display mouse cursor
        int     33h                         ;
        mov     eax,FALSE       ; return success

#exit:
        pop     es              ; restore regs
        ret                                 ; & exit

#buttons:
        lea     edx,b2msg
        jmp     short #err
#no_driver:
        lea     edx,drivmsg
        jmp     short #err
#not_real:
        lea     edx,realmsg
        jmp     short #err
```

```
#setup_err:
        lea     edx,setupmsg
        jmp     short #err
#bad_call:
        lea     edx,callmsg
#err:
        mov     ah,0                ; restore video mode
        mov     al,vid_mode                 ;
        int     10h                         ;
        mov     ah,9                ; output error msg
        int     21h                         ;
        mov     eax,TRUE            ; return error
        jmp     #exit                       ;

init endp

;*****************************************************
; cleanup - This routine resets the mouse driver to
;       set the interrupt call mask to 0 and hide
;       the cursor, and restores the original video
;       mode.
;*****************************************************
cleanup proc near
        mov     ax,0                ; reset mouse
        int     33h                         ;
        mov     ah,0                ; restore video mode
        mov     al,vid_mode                 ;
        int     10h                         ;
#exit:
        ret
cleanup endp

;*****************************************************
; print_ch(cond_msk)
; WORD cond_msk;
;
;       This routine is called from real mode
;       when either mouse button is pressed and we
;       are in character input mode.
;
;       If the left button was pressed, this proc
;       puts us in toggle mode by setting the
;       mouse handler routine to be tog_hndlr.
;
;       If the right button was pressed, this proc
;       outputs the current output character at
;       the specified cursor position.
;
;       This routine illustrates how to issue a
;       real mode interrupt with arbitrary segment
;       register values.
;
```

```
; Inputs:
;       1st stack param = mouse condition mask
;       CX = horizontal cursor coord
;       DX = vertical cursor coord
;
; Outputs:
;       AX = TRUE if error, FALSE if success
;       BX-DX = destroyed
;       High word of all 32-bit registers preserved
;*****************************************************
;
; Data structure passed to system call 2511h, issue
; real mode interrupt with arbitrary seg reg values
;
RMINT   struc
        RMI_INUM dw     ?       ; interrupt number
        RMI_DS  dw      ?       ; real mode DS
        RMI_ES  dw      ?       ; real mode ES
        RMI_FS  dw      ?       ; real mode FS
        RMI_GS  dw      ?       ; real mode GS
        RMI_EAX dd      ?       ; EAX value
        RMI_EDX dd      ?       ; EDX value
RMINT   ends

print_ch proc   far
;
; Stack frame
;
#COND_MSK equ   (word ptr 12[ebp])
#RMI    equ     (dword ptr [ebp - (size RMINT)])

        push    ebp             ; Set up stack frame
        mov     ebp,esp                 ;
        sub     esp,size RMINT  ; allocate local vars

        test    #COND_MSK,2     ; branch if left button
        jz      short #right            ; not pressed

;
; Left mouse button was pressed.  Issue a real mode
; interrupt to set up the mouse handler routine to
; be tog_hndlr.  The interrupt takes a function number
; in AX, an interrupt mask in CX, and the real mode
; addr of the handler in ES:DX.  We don't bother to
; initialize the values in the struct for DS,FS, or
; GS, and just let them get loaded with garbage.
;
        mov     ax,rm_seg       ; ES:DX = addr of
        mov     #RMI.RMI_ES,ax          ; handler
        lea     eax,tog_hndlr           ;
        mov     #RMI.RMI_EDX,eax        ;
        mov     cx,0Ah          ; interrupt mask
        mov     #RMI.RMI_EAX,12 ; EAX = function #
        mov     #RMI.RMI_INUM,33h ; interrupt number
        push    ds              ; issue real mode
        mov     ax,ss                   ; interrupt
        mov     ds,ax                   ;
        lea     edx,#RMI                ;
        mov     ax,2511h                ;
        int     21h                     ;
        pop     ds                      ;
        mov     ax,FALSE        ; return success
        jmp     short #exit             ;

#right:
;
; Right mouse button was pressed.  Set up ES to point
; to screen segment.
;
        mov     ax,001Ch
        mov     es,ax

;
; Get offset in screen segment of 1st character on
; the line the cursor is on
;
        shr     dx,3            ; convert pixel to line
        imul    dx,160          ; 160 bytes/line

;
; Add in the column number * 2 (2 bytes/char)
;
        shr     cx,2
        add     cx,dx
        mov     bx,cx

;
; Output the character at the cursor.
;
        mov     ah,7            ; normal attribute
        mov     al,outp_ch      ; character
        mov     word ptr es:[bx],ax ; output it

        mov     ax,FALSE        ; return success

#exit:
        add     esp,size RMINT  ; pop local vars
        pop     ebp             ; restore regs &
        ret                             ; exit
print_ch endp

pmcode  ends

page
```

```
;
; Real mode code
;
rmcode  segment

;**************************************************
; rm_setup(hndlr_seg)
; WORD hndlr_seg;
;
;       This routine is called from protected
;       mode.  It calls the mouse driver to set up
;       the specified address as the mouse interrupt
;       handler when either mouse button is pressed.
;       This call is made in real mode because
;       it passes a value in ES.
;
; Inputs:
;       1st stack param = segment addr of handler
;       DX = offset of handler
;
; Outputs:
;       AX = TRUE if error, FALSE if success
;       ES,BX-DX = destroyed
;       all other registers unchanged
;**************************************************
        assume  cs:rmcode,ds:dgroup
rm_setup proc   far
;
; Stack frame
;
#HNDLR_SEG equu  (word ptr 6[bp])

        push    bp              ; set up stack frame
        mov     bp,sp                   ;

        mov     ax,#HNDLR_SEG   ; ES:DX = addr of
        mov     es,ax                   ; handler
        mov     cx,0Ah          ; interrupt mask
        mov     ax,12           ; set up handler
        int     33h                     ;

        mov     ax,FALSE        ; return success

        pop     bp              ; restore regs &
        ret                             ; exit
rm_setup endp
```

```
;**************************************************
; ch_hndlr - This routine is invoked by the mouse
;       driver when either mouse button is pressed,
;       and when we are in character output mode.
;       This handler just calls a protected mode
;       routine to process the interrupt.  Since
;       this is a hardware interrupt handler, it is
;       obligated to preserve all 32-bit registers.
;
;       This routine demonstrates passing values
;       both on the stack and in registers to a
;       protected mode subroutine.
;**************************************************
        assume  nothing,cs:rmcode
ch_hndlr proc   far
        push    ds                  ; save regs
        push    eax                         ;

;
; To call a prot mode routine, all we do is push
; any params to the routine onto the stack, then
; push the FAR addr of the desired prot mode
; routine, then call the 386|DOS-Extender routine
; to switch modes and transfer control to the prot
; mode routine.
;
        push    cs                  ; set up data segment
        pop     ds                          ;
        assume  ds:dgroup                   ;

        push    ax                  ; param = cond. mask
        push    dword ptr 0         ; no seg reg param blk
        push    code_sel            ; prot mode addr of
        lea     eax,print_ch                ; display
        push    eax                         ; routine
        call    call_prot           ; call it
        add     sp,12               ; pop arguments
        cmp     ax,TRUE             ; branch if error
        je      short #err                  ;

#exit:
        pop     eax                 ; restore regs &
        pop     ds                          ; exit
        ret                                 ;

#err:
        lea     dx,chndlrmsg        ; output error msg
        mov     ah,9                        ;
        int     21h                         ;
        jmp     #exit
ch_hndlr endp
```

```
;****************************************************
; tog_hndlr - This routine is invoked by the mouse
;       driver when either mouse button is pressed,
;       and we are in toggle mode (the left mouse
;       button was the last one pressed).
;       If the left button has been pressed, we
;       set the done flag.  If the right button has
;       been pressed, we toggle the character that
;       gets output between '*' and ' ', and then
;       restore ch_hndlr as the mouse interrupt
;       handler.  Since this is a hardware
;       interrupt handler, it is obligated to
;       preserve all 32-bit registers.
;****************************************************
        assume  nothing,cs:rmcode
tog_hndlr proc  far
        push    ds                      ; save regs
        push    es                              ;

        push    cs                      ; set up data segment
        pop     ds                              ;
        assume  ds:dgroup                       ;

        test    ax,2                    ; branch if left button
        jz      short #right                    ; not pressed
        mov     donef,TRUE              ; set done flag & exit
        jmp     short #exit                     ;

#right:
        mov     al,' '                  ; assume char = ' '
        cmp     outp_ch,'*'             ; toggle output char
        je      short #space                    ;
        mov     al,'*'                          ;
#space:                                         ;
        mov     outp_ch,al                      ;
        mov     ax,cs                   ; Restore ch_hndlr
        mov     es,ax                           ; as mouse
        lea     dx,ch_hndlr                     ; handler
        mov     cx,0Ah                          ;
        mov     ax,12                           ;
        int     33h                             ;

#exit:
        pop     es                      ; restore regs &
        pop     ds                              ; exit
        ret                                     ;
tog_hndlr endp

rmcode  ends

        end main                        ; program entry point
```

The segment directives at the top of this program specify the desired segment order. The real-mode code and data must be placed at the beginning of the program when it is linked. The protected-mode data is placed immediately following the real-mode data, so that the two data segments can be grouped together. The ASSUME directive can then be used to tell the assembler that DS points to the data group, so that the assembler will permit references to data in both segments in the protected-mode code. This grouping of the data segments is not a requirement, but if it is not done, the assembler will require an ASSUME directive each time you need to reference data in a different data segment.

With the program organized in this manner, the -REALBREAK switch can be used at link time to specify how much of the program must be loaded in conventional memory. The program is built with the following commands:

```
386asm tail
386link tail -realbreak end_real
```

The program is then run by typing the following command:

```
run386 tail
```

The first thing the program does when it begins executing is call an initialization routine. This routine performs four functions:

- It sets the video mode, so that the program will know how to interpret the mouse coordinates. Note that segment 001Ch, used to access the screen memory, is automatically updated if necessary when the BIOS call is made to set the video mode.
- It calls system function 250Dh to obtain the real-mode address of the 386|DOS-Extender routine used to call from real mode to protected mode. It then saves this address in a global variable in the real-mode data segment, so that real-mode code can access it.
- It initializes the mouse driver.
- It uses system function 250Eh to call a real-mode initialization routine. Data is passed both on the stack and in a register to the real-mode routine. The real-mode routine just calls the mouse driver to set up the handler to be invoked when a mouse button is pressed.

Once the initialization is complete, the main program just sits in a loop waiting for the handler to set a flag saying the processing is complete so it can exit. The rest of the processing is done by the handler invoked each time a mouse button is pressed.

The program is always in one of two states: character mode or toggle mode. Character mode means that it will output a character each time the right mouse button is pressed. Toggle mode is entered by pressing the left mouse button.

When in toggle mode, pressing the left mouse button causes the done flag to be set (so the program will terminate). Pressing the right mouse button toggles the output character and re-enters character mode.

The program states are implemented by having two separate mouse handler routines, corresponding to the two program states. Both of these handlers are real-mode routines, because they are invoked by the mouse driver, which is a real-mode program.

The toggle mode handler routine is not particularly interesting, since it executes entirely in real mode. The character mode handler is more complex. It calls a protected-mode routine to actually write the character on the screen, using the 386 | DOS-Extender routine that calls through to protected mode. Data are passed to the protected-mode routine both in registers and on the stack.

Note that neither mouse handler has provisions to prevent being re-entered; this is unnecessary, since the mouse driver prevents it. Also note that the handlers don't need to preserve 16-bit registers, since that is also done by the driver. However, the driver does **not** preserve the 32-bit registers, because it is a real-mode program. Consequently, the handlers are responsible for saving and restoring any 32-bit registers that they modify.

The protected-mode routine that is called by the character mode handler performs two functions. If the right mouse button is pressed, it outputs a character at the current cursor position. If the left mouse button is pressed, it switches to toggle mode by setting up the toggle mode handler as the mouse handler routine.

For the left button function, the routine uses 386 | DOS-Extender system call 2511h to issue a real-mode interrupt (the mouse driver interrupt) with segment registers specified. This is necessary because the address of the handler is passed to the driver in the ES:DX registers. Note that this method is simpler than the one used in the initialization routine to accomplish the same result. There we called a real-mode routine whose only purpose was to call the mouse driver to set up a handler routine.

7.9.3 PTAIL2.ASM Microsoft Mouse Example

The PTAIL2 example program does the same thing as the TAIL.ASM example, but is implemented as separate protected mode (PTAIL2.ASM) and real mode (RTAIL2.ASM) programs. Like the GDEMO.C example, the protected mode program starts up first and then EXECs to the real mode program.

This sample program demonstrates mode switching using software interrupts rather than cross-mode procedure calls. Like the FAR pointer version of GDEMO.C, it also uses the buffer allocated with the -CALLBUFS switch to pass data between real- and protected-mode code.

Chapter 8

Compatibility

8.1 Direct Extended Memory Allocation

386 | DOS-Extender allows a protected-mode application program to allocate any direct extended memory that is not in use by other programs. At initialization time, 386 | DOS-Extender obtains the minimum available extended memory address by checking for memory that was allocated starting at one MB, with the VDISK extended memory allocation standard or with the technique used by the Microsoft RAMDRIVE program. 386 | DOS-Extender also obtains the top of extended memory by calling the BIOS system function INT 15h, function 88h. These memory allocation standards are described in detail in Chapter 5.

If you have programs installed on your system that use extended memory but do not use any of the standard methods for allocating extended memory, then it is necessary to limit the amount of extended memory that 386 | DOS-Extender will use to avoid conflict. This is done by determining which addresses in extended memory are used by the other program and then limiting the 386 | DOS-Extender use of extended memory with the -EXTLOW and -EXTHIGH command line switches.

There is a secondary problem with the VDISK memory allocation standard. The amount of memory allocated with this method is marked in two places: in a signature block pointed to by the INT 19h interrupt vector, and in a boot block maintained at one MB. Some programs that use this standard only update the allocation mark in one location, usually the signature block of the INT 19h vector. Also, under MS-DOS version 3.3, the resident portion of the print command takes over the INT 19h vector, which masks the VDISK signature block.

Because of these problems, 386 | DOS-Extender checks both VDISK allocation marks. If the two values are not the same, 386 | DOS-Extender prints out an error message showing what the two values are and refuses to run the application program. You should then check all the drivers in your CONFIG.SYS file to see which of them attempt to allocate extended memory, to determine whether the larger of the two numbers that 386 | DOS-Extender is finding is correct. If it is, you can use the -VDISK command line switch to tell 386 | DOS-Extender to proceed, using the larger of the two available numbers. If neither number is correct (this would be the case, for example, if you have a program that updates the VDISK signature block but not the boot block, and you then run the MS-DOS PRINT command under MS-DOS version 3.3), then you must use the -EXTLOW command line switch to give 386 | DOS-Extender the correct value.

8.2 Address Line 20 Enabling

All PC-compatible architectures have the capability of enabling or disabling address line 20 in hardware. When address line 20 is disabled, the physical addresses placed on the bus by the 80386 are truncated to 20 bits by external hardware. This is for compatibility with MS-DOS programs that rely on address wraparound at 1 MB; address line 20 is, therefore, typically disabled when MS-DOS is executing in real mode.

In order for protected-mode programs to run correctly, address line 20 must be enabled so that it is possible to access memory above one MB. By default, 386 | DOS-Extender enables address line 20 during initialization and leaves it enabled for the entire time the application program is executing. When the application program terminates, 386 | DOS-Extender returns address line 20 to whatever setting (enabled or disabled) it had originally.

The command line switch -A20 can be used to force 386 | DOS-Extender to enable address line 20 each time it switches to protected mode, and to disable it each time it switches to real mode. The only time this switch is needed is when a real-mode program that relies on address wraparound at one MB will be executing **while** the protected-mode program is executing. Examples include a terminate and stay resident program or a real-mode program to which the protected-mode program EXECs.

There is a penalty associated with the use of the -A20 switch. On the original PC AT hardware design, it took several milliseconds to enable or disable A20.

Thus, on systems with this design, using the -A20 switch adds several milliseconds to each mode switch.

However, all COMPAQ 386 machines manufactured after January 1987, the IBM PS/2 architecture, and the EISA architecture can switch the A20 state in a few microseconds.

8.3 Hardware Interrupt Conflicts

All PC-compatible architectures have hardware interrupts that conflict with exceptions that can be generated by the 80386 processor. 386 | DOS-Extender is capable of detecting whether an interrupt in the range 08h-0Fh resulted from a processor exception or a hardware interrupt.

For each protected-mode interrupt from 08h to 0Fh, the shadow IDT has two entries that can be read and written via separate system calls. (Please see sections 6.2 and 6.3.) One is for a hardware interrupt handler and the other is for a processor exception handler. The umbrella 386 | DOS-Extender handler for interrupts 8-F determines whether the source of the interrupt is a hardware interrupt or a processor interrupt. For hardware interrupts, control transfers to the hardware interrupt handler; otherwise, control transfers to the processor exception handler.

8.4 VCPI Interface

386 | DOS-Extender supports the Virtual Control Program Interface (VCPI) provided by EMS emulators. This interface was designed by Phar Lap Software, Inc. and Quarterdeck Office Systems to allow compatibility between EMS emulators and MS-DOS extenders. 386 | DOS-Extender can run in any environment that provides the VCPI interface. Copies of the interface specification can be obtained from Phar Lap Software, Inc. upon request.

Please see also Appendix G.

8.5 DPMI

DPMI (DOS Protected-Mode Interface) is an interface designed to allow MS-DOS extenders to run compatibly with multitasking operating systems, while allowing the multitasking host to virtualize the hardware, in particular the display. (Virtualizing hardware means that each active application program can behave as though it is the only user of the hardware devices on the computer.) Currently, DPMI is provided by Microsoft Windows 3.0. It appears likely that future versions of OS/2 and UNIX DOS boxes will also provide DPMI. Copies of the DPMI specification can be obtained from Intel Corp., order number 240743.

Version 3.0 of 386 | DOS-Extender is not compatible with DPMI. However, Phar Lap is committed to adding DPMI compatibility. Therefore, this manual includes guidelines for programmers who want to prepare for making their programs DPMI-compatible.

An important characteristic of DPMI is that it requires the application to run at a nonprivileged level. The actual privilege level is selected by the host operating system, and can be level 1, 2, or 3. Windows 3.0 runs applications at level 1. A DPMI-compatible application should therefore run at an unprivileged level by using the 386 | DOS-Extender switch setting -UNPRIV. The application should not assume it will run at any specific privilege level, however; please see sections 3.1, 3.2, and 4.1 for more information on segment selectors and privilege levels.

Not all switches and system calls provided by 386 | DOS-Extender can be supported under DPMI. This manual documents which features are not available under DPMI. Some features are available under some DPMI implementations but not others, because there are two versions of the DPMI interface — 0.9 and 1.0. Version 1.0 includes optional features that may not be supported in all DPMI hosts. To determine which DPMI capabilities are available (and therefore which 386 | DOS-Extender system calls are supported), an application can use the Get Configuration Information system call (function 2526h). Please see Appendix F for more information on writing DPMI-compatible applications.

8.6 XMS Drivers

XMS (eXpanded Memory Specification) is a Microsoft interface for allocating extended memory. The XMS interface is usually provided by a driver called HIMEM.SYS, and is a standard feature of both Windows 3.0 and MS-DOS 5.0.

386 | DOS-Extender is fully compatible with XMS.

8.7 Quarterdeck DESQview 386

DESQview 386 is a multitasking environment that runs on top of MS-DOS. 386 | DOS-Extender is capable of executing under DESQview 386, provided that the QEMM EMS emulator that is distributed with DESQview 386 is also installed.

DESQview relocates hardware interrupts from their default MS-DOS interrupt vectors. However, DESQview presents hardware interrupts to real-mode programs on the standard interrupt vectors, and 386 | DOS-Extender presents hardware interrupts to protected-mode programs on the standard interrupt vectors, so application programs can operate normally under DESQview. Please see section 6.1.2 and Appendix G for more information.

8.8 Windows 3.0

Windows 3.0 has three modes of operation:

- real mode — 8086–compatible
- standard mode — 286–compatible
- enhanced mode — 386–compatible; DPMI interface provided.

386 | DOS-Extender version 3.0 can run in real mode and standard mode, but **not** enhanced mode. When DPMI support is added to 386 | DOS-Extender, it will also be able to run in enhanced mode.

8.9 Memory Resident Programs

Memory resident programs, also referred to as terminate and stay resident (TSR) programs, are usually installed in memory when MS-DOS is initialized, through a line in the CONFIG.SYS or AUTOEXEC.BAT files. These programs frequently cause compatibility problems, because they take over interrupts. Typically, a TSR program takes over the hardware keyboard interrupt to enable it to "pop-up" at any time while another program is executing. Some TSR programs also make use of extended memory.

386 | DOS-Extender can execute compatibly with most memory resident programs. The sections below discuss some of the classes of memory resident programs that are available and some of the characteristics of these programs.

8.9.1 RAM Disks and Disk Cache Programs

RAM disks are programs that simulate a disk drive in memory. They are often used to hold files that are accessed frequently to improve access times. Disk cache programs cache data read from or written to the physical disk drives on a system, the purpose being to improve the performance of the system.

The only potential compatibility problem between 386 | DOS-Extender and either RAM disk or disk cache programs is their use of extended memory. 386 | DOS-Extender is compatible with all programs that follow either the VDISK standard, the BIOS extended memory size determination standard, or the technique used by the Microsoft RAMDRIVE program, for allocating extended memory. These standards are described in section 5.2.2. If memory resident programs that allocate extended memory, but do not follow one of these two standards, are used, the -EXTLOW and/or -EXTHIGH command line switches must be used to inform 386 | DOS-Extender how much extended memory is allocated to these programs.

8.9.2 EMS Emulators

The acronym EMS refers to the Lotus-Intel-Microsoft Expanded Memory Specification, established to enable standard real-mode MS-DOS applications to access more memory. The EMS specification was originally implemented in hardware.

However, as the 80286 and 80386 processors were introduced, able to address memory above one MB, a class of programs called EMS emulators became available. These programs use extended memory in place of add-on hardware to provide the extra memory. EMS emulators executing on the 80286 are slower than EMS hardware because they have to copy memory contents to and from extended memory.

On 80386 machines, however, EMS emulators are competitive with add-on hardware, because the 80386 can use the virtual 8086 (V86) mode of execution. This mode supports execution of standard 8086 programs, just as real mode does, but it also allows 80386 paging to be enabled. The EMS emulators can therefore use paging instead of physical copying to make memory available to real-mode programs.

Most 80386 EMS emulators execute in the V86 mode of the 80386. However, 386 | DOS-Extender cannot execute in V86 mode unless the EMS emulator has the VCPI interface documented in section 8.4.

VCPI is now an industry-wide standard, and is supported by most popular EMS emulators, including Microsoft EMM386, Quarterdeck QEMM-386, Qualitas 386MAX, and COMPAQ CEMM. If a program that does not provide the VCPI interface switches the 80386 into V86 mode, 386 | DOS-Extender will refuse to run. The only option then is to turn the EMS emulator off, which switches the 80386 processor back to real mode so that 386 | DOS-Extender can run.

8.9.3 Other Memory Resident Programs

Most other available memory resident programs run wholly in the real mode of the 80386. These programs generally take over one or more hardware interrupts to enable them to "pop-up" under certain conditions, such as a specific "hot key" being pressed. In general, 386 | DOS-Extender has no compatibility problems with this class of program, because 386 | DOS-Extender does not take over any hardware interrupts.

MS-DOS System Calls

MS-DOS calls are made with software interrupts 20h through 3Fh. Parameters are passed in registers. All registers, except those used to return results, are preserved across calls. This appendix lists all supported MS-DOS system calls.

Some interrupts perform more than one function, and one of the inputs (usually in register AH) is the function code for the desired function. Undocumented MS-DOS interrupts, and undocumented functions for multifunction interrupts, are passed through to MS-DOS with all general registers unmodified and with the DS and ES segment registers destroyed, as described in section 6.1 of this manual.

Many undocumented system calls do not use segment registers; these can be issued directly from protected mode, using the standard real mode calling conventions. To use an undocumented system call that uses segment registers:

☛ Use the -CALLBUFS switch to allocate a data buffer in conventional memory.

☛ At run time, call INT 21h function 250Dh to obtain real mode and protected mode pointers to the buffer allocated with -CALLBUFS.

☛ Use INT 21h function 2511h to issue the desired interrupt in real mode, with the appropriate segment register set to the real mode address of the conventional memory buffer.

The description of each system call also identifies in which version of MS-DOS the function was first supported (with MS-DOS 3.0 used as a baseline, since 386 | DOS-Extender cannot run under MS-DOS 1.x or 2.x). For example:

all	means the function is always executable under 386 | DOS-Extender (it was first supported in MS-DOS 3.0 or earlier).
3.3	means the function is available in MS-DOS 3.3 or later.
undoc	means the function is not officially documented by Microsoft. Please see reference 11 in the Preface for more information on undocumented DOS calls.

TABLE A-1
MS-DOS SYSTEM CALL SUMMARY

Interrupt Number	Function Number	Function Name	First MS-DOS Version
20h	none	Program Terminate (old form)	all
21h	00h	Program Terminate (old form)	all
21h	01h	Character Input with Echo	all
21h	02h	Character Output	all
21h	03h	Auxiliary Input	all
21h	04h	Auxiliary Output	all
21h	05h	Printer Output	all
21h	06h	Direct Console I/O	all
21h	07h	Unfiltered Character Input	all
21h	08h	Character Input	all
21h	09h	Output Character String	all
21h	0Ah	Buffered Input	all

(cont.)

Interrupt Number	Function Number	Function Name	First MS-DOS Version
21h	0Bh	Get Input Status	all
21h	0Ch	Flush Buffer and Input	all
21h	0Dh	Disk Reset	all
21h	0Eh	Select Default Disk	all
21h	19h	Get Default Disk	all
21h	1Ah	Set Disk Transfer Address	all
21h	1Bh	Get Default Disk Size Information	all
21h	1Ch	Get Disk Size Information	all
21h	26h	Create PSP	all
21h	2Ah	Get System Date	all
21h	2Bh	Set System Date	all
21h	2Ch	Get System Time	all
21h	2Dh	Set System Time	all
21h	2Eh	Set Verify Flag	all
21h	2Fh	Get Disk Transfer Address	all
21h	30h	Get MS-DOS and 386 I DOS-Extender Version Number	all
21h	31h	Terminate and Stay Resident	all
21h	32h	Get MS-DOS Disk Block	all undoc
21h	33h	Get or Set CTRL-BREAK Check Flag	all
21h	34h	Get Pointer to InDOS Flag	all undoc
21h	36h	Get Disk Free Space	all
21h	38h	Get or Set Country	all
21h	39h	Create Subdirectory	all
21h	3Ah	Delete Subdirectory	all

(cont.)

Interrupt Number	Function Number	Function Name	First MS-DOS Version
21h	3Bh	Set Current Directory	all
21h	3Ch	Create File	all
21h	3Dh	Open File	all
21h	3Eh	Close File	all
21h	3Fh	Read File or Device	all
21h	40h	Write File or Device	all
21h	41h	Delete File	all
21h	42h	Seek in File	all
21h	43h	Get or Set File Attributes	all
21h	44h	Device I/O Control	1
21h	45h	Duplicate Handle	all
21h	46h	Force Duplicate of Handle	all
21h	47h	Get Current Directory	all
21h	48h	Allocate Segment	all
21h	49h	Free Segment	all
21h	4Ah	Resize Segment	all
21h	4Bh	Execute Program	all
21h	4Ch	Terminate with Return Code	all
21h	4Dh	Get Child Return Code	all
21h	4Eh	Search for First Match	all
21h	4Fh	Search for Next Match	all
21h	50h	Set Program Segment Prefix (PSP)	all

(cont.)

1. Depends on input.

Interrupt Number	Function Number	Function Name	First MS-DOS Version
21h	51h	Get Program Segment Prefix (PSP)	all
21h	52h	Get List of Lists	all undoc
21h	54h	Get Verify Flag	all
21h	56h	Rename File	all
21h	57h	Get or Set File Date and Time	all
21h	58h	Get or Set Conventional Memory Allocation Strategy	all
21h	59h	Get Extended Error Information	all
21h	5Ah	Create Temporary File	all
21h	5Bh	Create File	all
21h	5Ch	File Locking	all
21h	5D06h	Get Address of DOS Swappable Data Area	all undoc
21h	5D0Bh	Get DOS Swappable Data Areas	4.0 undoc
21h	5E00h	Get Machine Name	3.1
21h	5E01h	Set Machine Name	3.1
21h	5E02h	Set Printer Setup String	3.1
21h	5E03h	Get Printer Setup String	3.1
21h	5F02h	Get Redirection List Entry	3.1
21h	5F03h	Redirect Device	3.1
21h	5F04h	Cancel Device Redirection	3.1
21h	60h	Resolve Path String	all undoc
21h	62h	Get Protected Mode Program Segment Prefix (PSP)	all
21h	65h	Get Extended Country Information	3.3
21h	66h	Get/Set Code Page	3.3

(cont.)

Interrupt Number	Function Number	Function Name	First MS-DOS Version
21h	67h	Set Handle Count	3.3
21h	68h	Commit File	3.3
21h	6C00h	Extended Open/Create File	4.0
22h	none	Address of Terminate Routine	all
23h	none	CTRL-C Handler	all
24h	none	Critical Error Handler	all
27h	none	Terminate and Stay Resident	all
2Fh	01h	Print Spooler	all
2Fh	0100h	Get Installed Status	all
2Fh	0101h	Print File	all
2Fh	0102h	Remove File from Queue	all
2Fh	0103h	Cancel all Files	all
2Fh	0104h	Hold for Status Read	all
2Fh	0105h	End Status Hold	all
2Fh	0600h	Check if ASSIGN Utility Installed	3.2
2Fh	1000h	Check if SHARE Module Loaded	3.2
2Fh	B700h	Check if APPEND Utility Installed	3.3
2Fh	B704h	Get APPEND Path Pointer	4.0
2Fh	B707h	Set APPEND Function State	4.0

INT 20h **Program Terminate**
ver: all

Input:	None
Output:	None

This outdated program terminate call is converted by 386 | DOS-Extender to the INT 21h function 4Ch terminate call, with a return code of FFh in register AL.

INT 21h **MS-DOS System Services**

Calls to INT 21h are used to request most MS-DOS system services. The number for the desired system function is passed in register AH. This is the preferred MS-DOS interface and is the one used by most well-behaved MS-DOS programs.

INT 21h **Program Terminate**
AH = 00h **ver: all**

Input:	None
Output:	None

This outdated program terminate call is converted by 386 | DOS-Extender to the INT 21h function 4Ch terminate call, with a return code of FFh in register AL.

INT 21h **Character Input with Echo**
AH = 01h **ver: all**

Input:	None	
Output:	AL	= 8-bit character

Reads a character from the standard input device (usually the keyboard), then echoes it to the standard output device (usually the display). If there is no character in the input buffer, this function waits for one to be available before it returns.

INT 21h **Character Output**
AH = 02h **ver: all**

Input:	DL	= 8-bit character
Output:	None	

Outputs the specified character to the standard output device (usually the display).

INT 21h **Auxiliary Input**
AH = 03h **ver: all**

Input:	None	
Output:	AL	= 8-bit character

Reads a character from the standard auxiliary device (usually serial communications port one).

INT 21h **Auxiliary Output**
AH = 04h **ver: all**

Input:	DL	= 8-bit character
Output:	None	

Outputs the specified character to the standard auxiliary device (usually serial communications port one).

INT 21h **Printer Output**
AH = 05h **ver: all**

Input:	DL	= 8-bit character
Output:	None	

Outputs a character to the standard list device (usually parallel port one).

INT 21h **Direct Console I/O**
AH = 06h **ver: all**

Input:	DL	= FFh, to read a character = 00h-FEh, to output the specified character code

Output: If input request and a character was ready:
  Zero Flag = clear
  AL = 8-bit character
If input request and no character available:
  Zero Flag = set
If outputting a character:
  None

Reads a character from the standard input device (usually the keyboard) without waiting, or writes any character other than FFh to the standard output device (usually the display).

INT 21h **Unfiltered Character Input**
AH = 07h **ver: all**

Input:	None	
Output:	AL	= 8-bit character

Reads a character from the standard input device (usually the keyboard) without echoing it. If no character is in the input buffer, it waits until one is available. Does not perform special processing of the CTRL-C or CTRL-BREAK characters.

INT 21h **Character Input**
AH = 08h **ver: all**

Input:	None	
Output:	AL	= 8-bit character

Reads a character from the standard input device (usually the keyboard) without echoing it. If no character is in the input buffer, it waits until one is available.

INT 21h **Output Character String**
AH = 09h **ver: all**

Input:	DS:EDX	= pointer to string to output
Output:	None	

Outputs the specified string, which must be terminated with the ASCII '$' character, to the standard output device (usually the screen).

INT 21h **Buffered Input**
AH = 0Ah **ver: all**

Input:	DS:EDX	= pointer to buffer to read data into
Output:	None	

Reads a line of input from the standard input device (usually the keyboard) and places it in the buffer. The first byte of the buffer must be initialized by the caller to specify the maximum number of bytes the buffer can hold. The second byte is filled in by MS-DOS with the number of characters actually read, not including the terminating carriage return. The data read are stored beginning at the third byte of the buffer.

INT 21h **Get Input Status**
AH = 0Bh **ver: all**

Input:	None	
Output:	AL	= 00h, if no character available
		= FFh, if character available

Returns the status of the standard input device (usually the keyboard).

INT 21h **Flush Buffer and Input**
AH = 0Ch **ver: all**

Input:	AL	= number of input function to invoke
	DS:EDX	= pointer to buffer, if input function 0Ah
Output:	AL	= character read, unless function 0Ah

Clears the input buffer and then performs the specified input function (01h, 06h, 07h, 08h, or 0Ah).

INT 21h **Disk Reset**
AH = 0Dh **ver: all**

Input:	None
Output:	None

Flushes all file buffers to disk.

INT 21h **Select Default Disk**
AH = 0Eh **ver: all**

Input:	DL	= drive number (0 = A:, 1 = B:, etc.)
OUPUTS:	AL	= number of logical drives on system

Selects the default disk to be used for disk I/O, and returns the total number of logical disk drives on the system.

INT 21h **File I/O Using FCBs**
AH = 0Fh through 17h

The MS-DOS file I/O functions that use FCBs (file control blocks) are not supported by 386 | DOS-Extender. The handle-based file I/O functions must be used instead.

INT 21h **Get Default Disk**
AH = 19h **ver: all**

Input:	None	
Output:	AL	= drive number (0 = A:, 1 = B:, etc.)

Returns the current default disk for disk I/O.

INT 21h **Set Disk Transfer Address**
AH = 1Ah **ver: all**

Input:	DS:EDX	= pointer to data buffer
Output:	None	

Sets the disk transfer address (DTA) to the specified buffer, which must be at least large enough for any function that may use it. The only functions that use it are 4Eh (find first matching file) and 4Fh (find next matching file). These functions use 43 bytes in the data buffer.

Because the program's DTA buffer may not be in conventional memory, 386 | DOS-Extender maintains a separate 128 byte DTA buffer. When a function call that uses the DTA buffer is made, 386 | DOS-Extender first calls MS-DOS to set the DTA buffer to its internal buffer. If the function passes data to MS-DOS in the buffer, it then copies data from the program's DTA buffer to the internal data buffer. It then makes the specified MS-DOS function call. Lastly, if the function returns data in the DTA buffer, the data are copied to the program's DTA buffer.

When the program is first loaded, the program's DTA buffer defaults to the 128-byte area beginning at offset 80h in the program's PSP.

INT 21h — Get Default Disk Size Information
AH = 1Bh **ver: all**

Input:	None	
Output:	AL	= number of sectors-cluster
	DS:EBX	= pointer to FAT identification byte
	CX	= size, in bytes, of physical sector
	DX	= number of clusters-disk

Returns size information for the current default disk drive.

INT 21h — Get Disk Size Information
AH = 1Ch **ver: all**

Input:	DL	= drive number (0 = current default, 1 = A:, 2 = B:, etc.)
Output:	AL	= number of sectors-cluster
	DS:EBX	= pointer to FAT identification byte
	CX	= size, in bytes, of physical sector
	DX	= number of clusters-disk

Returns size information for the specified disk drive.

INT 21h — File I/O Using FCBs
AH = 21h through 24h

The MS-DOS file I/O functions that use FCBs (file control blocks) are not supported by 386 | DOS-Extender. The handle-based file I/O functions must be used instead.

INT 21h **Set Interrupt Vector**
AH = 25h

The MS-DOS get and set interrupt vector functions are replaced by 386 | DOS-Extender system calls. See Appendix B for a list of these system calls.

INT 21h **Create PSP**
AH = 26h **ver: all**

Input:	DX	= segment paragraph address of new PSP
Output:	None	

This call copies the contents of the current PSP to the specified address and initializes the new PSP.

INT 21h **File I/O Using FCBs**
AH = 27h through 29h

The MS-DOS file I/O functions that use FCBs (file control blocks) are not supported by 386 | DOS-Extender. The handle-based file I/O functions must be used instead.

INT 21h **Get System Date**
AH = 2Ah **ver: all**

Input:	None	
Output:	AL	= day of week (0 = Sunday, 1 = Monday, etc.)
	CX	= year (1980 - 2099)
	DH	= month (1 - 12)
	DL	= day (1 - 31)

Returns the current system date.

INT 21h **Set System Date**
AH = 2Bh **ver: all**

Input:	CX	= year (1980 - 2099)
	DH	= month (1 - 12)
	DL	= day (1 - 31)
Output:	AL	= 00h, if date set
		= FFh, if invalid date

Sets the system date to the specified date.

INT 21h AH = 2Ch — Get System Time

ver: all

Input:	None	
Output:	CH	= hour (0 - 23)
	CL	= minutes (0 - 59)
	DH	= seconds (0 - 59)
	DL	= hundredths of seconds (0 - 99)

Returns the current system time.

INT 21h AH = 2Dh — Set System Time

ver: all

Input:	CH	= hour (0 - 23)
	CL	= minutes (0 - 59)
	DH	= seconds (0 - 59)
	D	= hundredths of seconds (0 - 99)
Output:	AL	= 00h, if time set
		= FFh, if invalid time

Sets the system time to the specified time.

INT 21h **Set Verify Flag**
AH = 2Eh **ver: all**

Input:	AL	= 00h, to turn off verify flag
		= 01h, to turn on verify flag
Output:	None	

Sets the system verify flag (used for automatic verification of data written to disk, at the expense of performance) as specified.

INT 21h **Get Disk Transfer Address**
AH = 2Fh **ver: all**

Input:	None	
Output:	ES:EBX	= pointer to DTA buffer

Returns a pointer to the current disk transfer address (DTA) buffer for the program.

INT 21h **Get MS-DOS and 386 | DOS-Extender Version Number**
AH = 30h **ver: all**

Input:	EBX	= "PHAR" to get 386 | DOS-Extender version in addition to MS-DOS version
		= any other value to get only MS-DOS version

Output:	AL	= MS-DOS major version number, in binary
	AH	= MS-DOS minor version number, in binary
	EAX <16-31>	= "DX" (for 386 | DOS-Extender)
	If 386 | DOS-Extender version requested:	
	BL	= 386 | DOS-Extender major version number, in ASCII
	BH	= 386 | DOS-Extender minor version number, in ASCII
	EBX <16-23>	= 386 | DOS-Extender version letter subcode, in ASCII, or ASCII space
	EBX <24-31>	= ASCII space
	ECX	= environment
		= "DOS" if MS-DOS only
		= "DESQ" if Quarterdeck DESQview 386
		= "VCPI" if a Virtual 8086 mode control program (such as Quarterdeck QEMM) that supports the VCPI interface
	EDX	= 0

This function returns the MS-DOS version number, plus a signature identifying the presence of 386 | DOS-Extender in the high word of register EAX. In addition, if the 386 | DOS-Extender version number is requested, its version number is returned in EBX, additional information about the environment is returned in ECX, and EDX is reserved for future expansion. The 386 | DOS-Extender version number is returned in ASCII and is formatted so that the contents of the EBX register can be stored in memory and then manipulated as an ASCII string. For example, version 3.0 of 386 | DOS-Extender returns the string "30 " in EBX. Version letter subcodes are used only for internal incremental releases of 386 | DOS-Extender.

Please see also 386 | DOS-Extender system call 2526h, Get Configuration Information.

INT 21h **Terminate and Stay Resident**
AH = 31h **ver: all**

Input:	AL	= return code
Output:	None	

The terminate and stay resident function exits back to MS-DOS, while still keeping the program and 386 | DOS-Extender in memory. Any open files remain open. TSR programs must leave free as much conventional memory and extended memory as possible, so that memory is available to run other programs. TSR programs are discussed in more detail in Section 7.8.

INT 21h **Get MS-DOS Disk Block**
AH = 32h **ver: all undoc**

Input: DL = drive number (0 = default, 1 = A:, etc.)

Output:
If drive exists:
  AL = 00h
  DS:EBX = pointer to MS-DOS disk block
If invalid drive number:
  AL = FFh

Returns a pointer to the MS-DOS disk block for the specified drive. The disk block contains the following information:

Offset	Size	Description
00h	BYTE	Drive number (0 = A:, 1 = B:, etc.)
01h	BYTE	Unit within driver
02h	WORD	Number of bytes/sector
04h	BYTE	Number of sectors/cluster - 1
05h	BYTE	Cluster to sector shift
06h	WORD	Number of reserved sectors
08h	BYTE	Number of file allocation table entries
09h	WORD	Number of root directory entries
0Bh	WORD	Sector number of cluster two
0Dh	WORD	Number of clusters + 1
0Fh	BYTE	Sectors for FAT
10h	WORD	Sector number of directory
12h	DWORD	Real-mode address of device header
16h	BYTE	Media descriptor byte
17h	BYTE	Zero if disk has been accessed
18h	DWORD	Real-mode address of next MS-DOS disk block

INT 21h **Get or Set CTRL-BREAK Check Flag**
AH = 33h **ver: all**

Input: If getting flag:
AL = 00h
If setting flag:
AL = 01h
DL = 00h, to turn CTRL-BREAK checking off
= 01h, to turn CTRL-BREAK checking on

Output: DL = 00h, if CTRL-BREAK checking off
= 01h, if CTRL-BREAK checking on

Gets or sets the current status of the MS-DOS CTRL-BREAK checking flag.

INT 21h **Get Pointer to InDOS Flag**
Ah = 34h **ver: all undoc**

Input:	None	
Output:	ES:EBX	= pointer to 1-byte MS-DOS critical section flag

Returns a pointer to a one-byte flag which is nonzero when code within MS-DOS is being executed.

INT 21h **Get Interrupt Vector**
AH = 35h

The MS-DOS get and set interrupt vector functions are replaced by 386 | DOS-Extender system calls. See Appendix B for a list of these system calls.

INT 21h **Get Disk Free Space**
AH = 36h **ver: all**

Input:	DL	= drive number (0 = current default, 1 = A:, 2 = B:, etc.)

Output: If invalid drive number:
  AX = FFFFh
If valid disk drive:
  AX = number of sectors/cluster
  BX = number of free disk clusters
  CX = number of bytes/sector
  DX = number of clusters/disk

Returns disk size information and the amount of free space on the disk.

INT 21h **Get or Set Country**
AH = 38h **ver: all**

```
Input:       AL      = country select
                     = 00h, for current country
                     = 01h - FEh, for country code < 255
                     = FFh, if country code in BX
             BX      = country code, if >= 255
             If getting current country information:
               DS:EDX  = pointer to 34-byte buffer
             If setting current country:
               EDX     = FFFFFFFFh
```

```
Output:      If success:
               Carry flag   = clear
               BX           = country code
               If getting current country information:
                 Buffer at DS:EDX filled in
             If failure:
               Carry flag   = set
               AX           = error code
                            = 2, if invalid country code
```

Gets or sets the current country used by MS-DOS. See Reference 3 for a description of the format of the data stored in the buffer.

INT 21h **Create Subdirectory**
AH = 39h **ver: all**

Input:	DS:EDX	= pointer to zero-terminated path name

```
Output:     If success:
              Carry flag   = clear
            If failure:
              Carry flag   = set
              AX           = error code
                           = 3, if path not found
                           = 5, if access denied
```

Creates a subdirectory in the specified location.

INT 21h **Delete Subdirectory**
AH = 3Ah **ver: all**

Input:	DS:EDX	= pointer to zero-terminated path name

```
Output:     If success:
              Carry flag   = clear
            If failure:
              Carry flag   = set
              AX           = error code
                           = 3, if path not found
                           = 5, if access denied or directory not empty
                           = 16, if current directory
```

Deletes the specified subdirectory, which must be empty and must not be the current default directory.

INT 21h AH = 3Bh — Set Current Directory

ver: all

Input:	DS:EDX	= pointer to zero-terminated path name

Output:	If success:	
	Carry flag	= clear
	If failure:	
	Carry flag	= set
	AX	= error code
		= 3, if path not found

Sets the current default directory to the specified disk drive and path.

INT 21h AH = 3Ch — Create File

ver: all

Input:	CX	= file attribute
		= 0, if normal file
		= 1, if read only
		= 2, if hidden
		= 4, if system file
	DS:EDX	= pointer to zero-terminated file name

Output:	If success:	
	Carry flag	= clear
	AX	= file handle
	If failure:	
	Carry flag	= set
	AX	= error code
		= 3, if path not found
		= 4, if too many open files
		= 5, if access denied

Creates a file in the current default directory or the specified directory, and returns a file handle to be used in subsequent file I/O calls. If the file already exists, it is opened and truncated to zero bytes.

INT 21h **Open File**
AH = 3Dh **ver: all**

Input:	AL	= access and file sharing (see Reference 3)
	DS:EDX	= pointer to zero-terminated file name

```
Output:     If success:
              Carry flag   = clear
              AX           = file handle
            If failure:
              Carry flag   = set
              AX           = error code
                           = 1, if invalid file sharing code
                           = 2, if file not found
                           = 3, if path not found
                           = 4, if too many open files
                           = 5, if access denied
                           = 12, if invalid file access code
```

Opens the specified file in the current default directory, or in the specified directory, and returns a file handle to be used in subsequent file I/O calls.

INT 21h AH = 3Eh — Close File

ver: all

Input:	BX	= file handle

Output:	If success:	
	Carry flag	= clear
	If failure:	
	Carry flag	= set
	AX	= error code
		= 6, if invalid handle

Closes the file previously opened under the specified file handle.

INT 21h AH = 3Fh — Read File or Device

ver: all

Input:	BX	= file handle
	ECX	= number of bytes to read
	DS:EDX	= pointer to buffer to read data into

Output:	If success:	
	Carry flag	= clear
	EAX	= number of bytes actually read
	If failure:	
	Carry flag	= set
	AX	= error code
		= 5, if access denied
		= 6, if invalid handle

Reads data from the file or device previously opened under the specified file handle. If end of file is encountered during the read, less data than requested may be returned.

INT 21h	**Write File or Device**
AH = 40h	**ver: all**

Input:	BX	= file handle
	ECX	= number of bytes to write
	DS:EDX	= pointer to buffer containing data to write

```
Output:     If success:
              Carry flag   = clear
              EAX          = number of bytes actually written
            If failure:
              Carry flag   = set
              AX           = error code
                           = 5, if access denied
                           = 6, if invalid handle
```

Writes data to the file or device previously opened under the specified file handle. If the device on which the file is located is full, less data than requested may actually be written.

INT 21h	**Delete File**
AH = 41h	**ver: all**

Input:	DS:EDX	= pointer to zero-terminated file name

```
Output:     If success:
              Carry flag   = clear
            If failure:
              Carry flag   = set
                AX         = error code
                           = 2, if file not found
                           = 5, if access denied
```

The specified file is deleted.

INT 21h **Seek in File**
AH = 42h **ver: all**

Input:	AL	= offset origin
		= 00h, if offset from beginning of file
		= 01h, if offset from current file pointer
		= 02h, if offset from end of file
	BX	= file handle
	CX	= high word of offset value
	DX	= low word of offset value

Output:	If success:	
	Carry flag	= clear
	DX	= high word of new file pointer
	AX	= low word of new file pointer
	If failure:	
	Carry flag	= set
	AX	= error code
		= 1, if invalid origin code in AL
		= 6, if invalid handle

Positions the file read-write pointer. The doubleword offset value is signed.

INT 21h **Get or Set File Attributes**
AH = 43h **ver: all**

Input:	AL	= 00h, if getting file attribute
		= 01h, if setting file attribute
	DS:EDX	= pointer to zero-terminated file name
	If setting file attribute:	
	CX	= new attribute
	bit 0	= read only
	bit 1	= hidden
	bit 2	= system
	bit 5	= archive

Output:	If success:	
	Carry flag	= clear
	If getting file attribute:	
	CX	= attribute
	If failure:	
	Carry flag	= set
	AX	= error code
		= 1, if invalid function code in AL
		= 2, if file not found
		= 3, if path not found
		= 5, if can't change attribute

Reads or modifies the attribute of the specified file.

INT 21h **Device I/O Control**
AH = 44h **ver: see each input**

Input:

If getting device information: ver: all

AL	= 00h
BX	= file or device handle

If setting device information: ver: all

AL	= 01h
BX	= file or device handle
DX	= device information

If reading from character device control channel: ver: all

AL	= 02h
BX	= device handle
CX	= number of bytes to read
DS:EDX	= pointer to data buffer

If writing to character device control channel: ver: all

AL	= 03h
BX	= device handle
CX	= number of bytes to write
DS:EDX	= pointer to data buffer

If reading from block device control channel: ver: all

AL	= 04h
BL	= drive number (0 = default, 1 = A:, etc.)
CX	= number of bytes to read
DS:EDX	= pointer to data buffer

If writing to block device control channel: ver: all

AL	= 05h
BL	= drive number (0 = default, 1 = A:, etc.)
CX	= number of bytes to write
DS:EDX	= pointer to data buffer

If getting input status: ver: all

AL	= 06h
BX	= file or device handle

If getting output status: ver: all

AL	= 07h
BX	= file or device handle

If testing whether block device is changeable: ver: all

AL	= 08h
BL	= drive number (0 = default, 1 = A:, etc.)

If testing whether drive local or remote: ver: 3.1

AL	= 09h
BL	= drive number (0 = default, 1 = A:, etc.)

If testing whether handle local or remote: ver: 3.1

AL	= 0Ah
BX	= device handle

If modifying sharing retry count: ver: 3.1

AL	= 0B
CX	= number of delay loops
DX	= number of retries

If set iteration count for character device: ver: 3.2

AL	= 0Ch
CL	= 45h
BX	= handle
CH	= category code
DS:EDX	= pointer to WORD iteration count

If select code page for character device: ver: 3.3

AL	= 0Ch
CL	= 4Ah
BX	= handle
CH	= category code
DS:EDX	= pointer to buffer of length 2+n bytes, where n is first WORD in buffer.

If start code page preparation for character device: ver: 3.3

AL	= 0Ch
CL	= 4Ch
BX	= handle
CH	= category code
DS:EDX	= pointer to buffer of length 4+n bytes, where n is second WORD in buffer

If end code page preparation for character device: ver: 3.3

AL	= 0Ch
CL	= 4Dh
BX	= handle
CH	= category code
DS:EDX	= pointer to buffer of length 2+n bytes, where n is second WORD in buffer

If set display/information for character device: ver: 4.0

AL = 0Ch
CL = 5Fh
BX = handle
CH = category code
DS:EDX = pointer to buffer of length 4+n bytes, where n is second WORD in buffer

If get iteration count for character device: ver: 3.2

AL = 0Ch
CL = 65h
BX = handle
CH = category code
DS:EDX = pointer to 2-byte buffer

If query selected code page for character device: ver: 3.3

AL = 0Ch
CL = 6Ah
BX = handle
CH = category code
DS:EDX = pointer to buffer of length 2+n bytes, where n is first WORD in buffer

If query prepare list for character device: ver: 3.3

AL = 0Ch
CL = 6Bh
BX = handle
CH = category code
DS:EDX = pointer to buffer of length 2+n bytes, where n is first WORD in buffer

If get display information for character device: ver: 4.0

AL = 0Ch
CL = 7Fh
BX = handle
CH = category code
DS:EDX = pointer to buffer of length 4+n bytes, where n is second WORD in buffer.

If get logical drive map: ver: 3.2

AL = 0Eh
BL = drive number (0 = default, 1=A:, etc.)

If get logical drive map: ver: 3.2

AL = 0Fh
BL = drive number (0 = default, 1=A:, etc.)

```
Output:     If success:
              Carry flag  = clear
              Function 00h:
                DX              = device information
              Functions 02h - 05h:
                AX              = number of bytes transferred
              Functions 06h and 07h:
                AL              = 00h, if device not ready
                                = FFh, if device ready
              Function 08h:
                AX              = 0, if removable media
                                = 1, if fixed media
              Functions 09h and 0Ah:
                DX              = device information
              Function 0Eh:
                AL              = mapping code
            If failure:
              Carry flag  = set
              AX          = error code
                          = 1, if invalid function in AL
                          = 5, if access denied
                          = 6, if invalid handle
                          = 13, if invalid data
                          = 15, if invalid drive number
```

This function is used to get or modify device driver parameters. The formats of the device information word, category codes, and buffer structures are specified in Reference 3 in the Preface.

INT 21h AH = 45h — Duplicate Handle

ver: all

Input:	BX	= file or device handle

Output:

If success:

Carry flag	= clear
AX	= new handle

If failure:

Carry flag	= set
AX	= error code
	= 4, if too many open files
	= 6, if invalid handle

Creates a new handle that refers to the same file or device as the specified handle.

INT 21h AH = 46h — Force Duplicate of Handle

ver: all

Input:	BX	= handle to duplicate
	CX	= new handle

Output:

If success:

Carry flag	= clear

If failure:

Carry flag	= set
AX	= error code
	= 4, if too many open files
	= 6, if invalid handle

Forces the new handle to refer to the same file or device as the original handle. If the new handle refers to an open file, the file is closed before the handle is turned into a duplicate.

INT 21h **Get Current Directory**
AH = 47h **ver: all**

Input:	DL	= drive number (0 = default, 1 = A:, etc.)
	DS:ESI	= pointer to 64-byte buffer in which to store directory name

Output:	If success:	
	Carry flag	= clear
	If failure:	
	Carry flag	= set
	AX	= error code
		= 15, if invalid drive number

Returns the zero-terminated name of the current default directory on the specified disk drive.

INT 21h **Allocate Segment**
AH = 48h **ver: all**

Input:	EBX	= number of 4 KB memory pages to allocate

Output:	If success:	
	Carry flag	= clear
	AX	= segment selector of allocated segment
	If failure:	
	Carry flag	= set
	EAX	= error code
		= 8, if insufficient memory or LDT is maximum size
	EBX	= number of free pages of physical memory available

Allocates a segment descriptor in the local descriptor table (LDT), and returns the segment selector used to reference the segment. The segment is always created as a writable expand-up data segment. A memory allocation of zero pages is explicitly allowed, in order to allocate an LDT descriptor which the program can then modify to suit its own needs.

Use the Get Memory Statistics call (function 2520h) to obtain more detailed information about available physical memory.

INT 21h | **Free Segment**
AH = 49h | **ver: all**

Input:	ES = segment selector of segment to free

```
Output:   If success:
            Carry flag   = clear
            ES           = 0000h (the null segment selector)
          If failure:
            Carry flag   = set
            EAX          = error code
                         = 9, if invalid segment selector
```

Frees the specified segment, and frees all of the memory pages allocated to the segment. The segment must be in the local descriptor table (LDT) and may not be one of the hardwired segments set up by 386 | DOS-Extender, with the exception of the original code and data segments.

If the code or data segment is freed, the program must have first moved its stack to another segment and be executing out of another segment, or the system will crash.

If a segment that has aliases is freed, only the segment descriptor is freed. Memory allocated to the segment is not freed until all of the aliases for the segment are freed. Segments 000Ch and 0014h (the program code and data segments) are automatically aliased. Additional aliases can be created with the Alias Segment Descriptor call (function 2513h).

INT 21h — Resize Segment
AH = 4Ah — **ver: all**

Input:	ES	= segment selector of segment to be modified
	EBX	= new requested segment size, in four-KB pages

```
Output:     If success:
              Carry flag   = clear
            If failure:
              Carry flag   = set
              EAX          = error code
                           = 8, if insufficient memory
                           = 9, if invalid segment selector
                           = 130, if not supported under this DPMI
                               implementation
              EBX          = number of free pages of physical memory
                               available
```

Modifies the amount of memory allocated to a segment and changes the segment limit in the LDT. The segment must be in the local descriptor table (LDT) and may not be one of the hardwired segments set up by 386 | DOS-Extender, with the exception of the original code or data segments. All segment registers are reloaded by 386 | DOS-Extender after the modification is made, to make sure the new segment limit will be correctly recognized.

This function is useful for releasing memory that a program does not need, so that it is available for other purposes. By default, programs are given all the memory in the machine when they are loaded. Well-behaved programs, therefore, release memory they do not need during initialization. Information on the program's initial size can be found in the PSP; see section 3.3.

If a segment that has aliases is modified, the descriptors for all of the aliased segments are automatically updated. Segments 000Ch and 0014h are always aliased. Additional aliases can be created with the Alias Segment Descriptor call (function 2513h).

If executing under DPMI, attempting to modify a segment with mapped physical memory created by function 250Ah will return error 130 unless DPMI Device Mapping capability (please see Appendix F) is available.

If the segment size is increased with this call, the linear base address of the segment in the LDT segment descriptor may change. This side effect is invisible to the application program. A moved segment handler can be installed with function 2518h; the installed handler is called whenever a segment linear base address changes.

INT 21h **Execute Program**
AH = 4Bh **ver: all**

Input:	AL	= 00h, if loading and executing program
		= all other values are invalid
	ES:EBX	= pointer to parameter block
	DS:EDX	= pointer to zero-terminated program file name

```
Output:     If success:
              Carry flag    = clear
              All registers unchanged
            If failure:
              Carry flag    = set
              EAX           = error code
                            = 1, if function code in AL is invalid
                            = 2, if file not found or path invalid
                            = 5, if access denied
                            = 8, if insufficient memory to load program
                            = 10, if environment invalid
                            = 11, if invalid file format
              All other registers unchanged
```

This function is used to load and execute other real-mode or protected-mode programs. Sufficient memory must have been left free for the child program to be loaded; see section 3.8.

The format of the parameter block pointed to by ES:EBX is:

Offset	Size	Description
00h	DWORD	Offset of environment string
04h	WORD	Segment selector of environment string
06h	DWORD	Offset of command tail string
0Ah	WORD	Segment selector of command tail string

If a null (zero) segment selector is passed for the environment string, the parent's environment block is duplicated for the child. Otherwise, the environment string is a sequence of zero-terminated strings, with the whole block terminated by an extra zero byte.

The command tail string consists of a count byte, followed by an ASCII string terminated with a carriage return. The count value does not include the count byte itself or the terminating carriage return — it only accounts for the number of bytes in the ASCII string.

INT 21h function 25C3h can also be used to EXEC another program. The 4Bh EXEC call performs the following two operations which are **not** done by the 25C3h call:

- It disables A20 (address line 20) if it was disabled when the program started running. Note, however, if a switch back to protected mode occurs before the EXEC terminates (because your application has installed a protected mode interrupt handler with system call 2506h, and the interrupt is issued), A20 is automatically re-enabled and stays enabled for the remainder of the EXEC.
- When running with 386|VMM, it flushes the page swap file (updates its directory entry) before performing the EXEC.

Most applications should use the 4Bh form of EXEC. The 25C3h call exists primarily for historical reasons.

INT 21h **Terminate with Return Code**
AH = 4Ch **ver: all**

Input:	AL	= return code
Output:	None	

Terminates the program, returning control to MS-DOS or the parent program. All open files are closed and the file buffers flushed to disk before terminating the program.

INT 21h **Get Child Return Code**
AH = 4Dh **ver: all**

Input:	None	
Output:	AH	= exit code
		= 00h, if normal exit
		= 01h, if termination by CTRL-C
		= 02h, if critical device error
		= 03h, if terminate and stay resident
	AL	= return code passed by child

After successful completion of an EXEC call (function 4Bh), this call can be used to obtain the return code of the child program and the type of exit performed by the child.

INT 21h AH = 4Eh — Search for First Match — ver: all

Input:	CX	= file attributes
	DS:EDX	= pointer to zero-terminated file path

Output:	If success:	
	Carry flag	= clear
	If failure:	
	Carry flag	= set
	AX	= error code
		= 2, if path invalid
		= 18, if no matching directory entry

Returns the first file in the specified directory which matches the file name and has a subset of the specified file attributes. The "*" and "?" wildcard characters may be used in the file name portion of the file path. The result is stored in the program's DTA buffer in the following format:

Offset	Size	Description
00h	21 BYTEs	Reserved by MS-DOS
15h	BYTE	File attribute
16h	WORD	File time
18h	WORD	File date
1Ah	DWORD	File size, in bytes
1Eh	13 BYTEs	Zero-terminated file name

INT 21h **Search for Next Match**
AH = 4Fh **ver: all**

Input:	None	
Output:	If success:	
	Carry flag	= clear
	If failure:	
	Carry flag	= set
	AX	= error code
		= 18, if no matching directory entry

When this call is made, the DTA buffer must contain the results of a previous call to function 4Eh or 4Fh. The result is stored in the DTA buffer in the format described for function call 4Eh.

INT 21h **Set Program Segment Prefix**
AH = 50h **ver: all**

Input:	BX	= real-mode paragraph address of PSP
Output:	None	

Instructs MS-DOS to set the current program segment prefix to the PSP at the specified memory address. This call should not normally be used by application programs, except mixed real- and protected-mode programs as described in section 7.2.4.

INT 21h **Get Program Segment Prefix**
AH = 51h **ver: all**

Input:	None	
Output:	BX	= real-mode paragraph address of PSP

Returns the real-mode segment paragraph address of the PSP. See section 7.2.4.

INT 21h **Get List of Lists**
AH = 52h **ver: all undoc**

Input:	None	
Output:	ES:EBX	= pointer to MS-DOS list of lists

The returned pointer is the address of a data structure within MS-DOS. Note that any FAR pointers in the data structure are real mode FAR pointers.

INT 21h **Get Verify Flag**
AH = 54h **ver: all**

Input:	None	
Output:	AL	= 00h, if verify off
		= 01h, if verify on

Returns the current state of the system flag used to enable automatic verification of data written to disk.

INT 21h **Rename File**
AH = 56h **ver: all**

Input:	DS:EDX	= pointer to current zero-terminated file name
	ES:EDI	= pointer to new zero-terminated file name

Output:	If success:		
		Carry flag	= clear
	If failure		
		Carry flag	= set
		AX	= error code
			= 2, if file not found
			= 3, if path not found
			= 5, if access denied
			= 17, if not same device

Renames the file and optionally moves it to a different directory on the same disk.

INT 21h **Get or Set File Date and Time**
AH = 57h **ver: all**

Input:	BX		= file handle
	If getting date and time:		
		AL	= 00h
	If setting date and time:		
		AL	= 01h
		CX	= time
		DX	= date

Output:	If success:	
	Carry flag	= clear
	If getting date and time:	
	CX	= time
	DX	= date
	If failure:	
	AX	= error code
		= 1, if invalid function code in AL
		= 6, if invalid handle

Returns or modifies the date and time of the specified file. The format of the time byte is:

bits 0-4	= number of two-second increments
bits 5-10	= minutes
bits 11-15	= hours

The format of the date byte is:

bits 0-4	= day
bits 5-8	= month
bits 9-15	= year (relative to 1980)

INT 21h AH = 58h — Get or Set Conventional Memory Allocation Strategy

ver: all

```
Input:       If getting strategy:
               AL   = 00h
             If setting strategy:
               AL   = 01h
               BX   = strategy code
                    = 0, if first fit
                    = 1, if best fit
                    = 2, if last fit
```

```
Output:      If success:
               Carry flag   = clear
               If getting strategy:
                 AX              = strategy code
             If failure:
               Carry flag   = set
               AX           = error code
                            = 1, if invalid function code in AL
```

This function gets or sets the allocation strategy used by MS-DOS to allocate conventional memory. This affects memory allocation done with 386 | DOS-Extender system calls 25C0h - 25C2h. This allocation strategy is unrelated to memory allocation done with INT 21h functions 48h - 4Ah.

INT 21h **Get Extended Error Information**
AH = 59h **ver: all**

Input:	BX	= 0

Output:	AX	= extended error code
	BH	= error class
	BL	= recommended action
	CH	= error locus
	CL, DX, SI, DI, BP, DS, ES	= destroyed
	If AX = 22h:	
	ES:EDI	= pointer to zero-terminated name of disk volume to insert

Returns extended error information from the previous MS-DOS system call. See Reference 3 for error codes.

INT 21h **Create Temporary File**
AH = 5Ah **ver: all**

Input:	CX	= file attribute
	DS:EDX	= pointer to zero-terminated path

Output:	If success:	
	Carry flag	= clear
	AX	= file handle
	DS:EDX	= file name appended to path name
	If failure:	
	Carry flag	= set
	AX	= error code
		= 3, if path not found
		= 5, if access denied

Creates a temporary file in the specified directory and returns the file name. The buffer pointed to by DS:EDX must have 13 bytes of space following the end of the path name.

INT 21h | **Create File**
AH = 5Bh | **ver: all**

Input:	CX	= file attribute
	DS:EDX	= pointer to zero-terminated file name

Output:	If success:	
	Carry flag	= clear
	AX	= file handle
	If failure:	
	Carry flag	= set
	AX	= error code
		= 3, if path not found
		= 4, if too many open files
		= 5, if access denied
		= 80, if file already exists

Creates the specified file in the specified or default directory. See Reference 3 for information on file attributes.

INT 21h **File Locking**
AH = 5Ch **ver: all**

Input:	AL	= 00h, if locking file region
		= 01h, if unlocking file region
	BX	= file handle
	CX	= high word of region offset
	DX	= low word of region offset
	SI	= high word of region length
	DI	= low word of region length

Output:	If success:	
	Carry flag	= clear
	If failure:	
	Carry flag	= set
	AX	= error code
		= 1, if invalid function code in AL
		= 6, if invalid handle
		= 33, if all or part of region already locked

This function is used in a multitasking or networking environment to lock or unlock a region of a file. The offset and length of the region to be locked are both in bytes. The file-sharing module (SHARE.EXE) must be loaded to use this function.

INT 21h **Get Address of DOS Swappable Data Area**
AH = 5D06h **ver: all**

Input:	None

Output: If success:

Carry flag	= clear
DS:ESI	= address of nonreentrant data area
CX	= size, in bytes, of area to swap while in MS-DOS
DX	= size, in bytes, of area to always swap

If failure:

Carry flag	= set
AX	= error code

Returns the address and size of the data area to swap to permit reentering MS-DOS.

INT 21h **Get DOS Swappable Data Areas**
AH = 5D0Bh **ver: 4.0 undoc**

Input:	None

Output: If success:

Carry flag	= clear
DS:ESI	= address of list of swappable data areas

If failure:

Carry flag	= set

The returned pointer is the address of a data structure within MS-DOS. Note that any FAR pointers in the data structure are real mode FAR pointers.

INT 21h AH = 5E00h — Get Machine Name — ver: 3.1

Input:	DS:EDX	= pointer to 16-byte buffer	
Output:	If success:		
	Carry flag	= clear	
	CH	= 0, if name not defined	
		= nonzero, if name defined	
	CL	= NETBIOS name number	
	ASCIIZ name string stored in buffer at DS:EDX		
	If failure:		
	Carry flag	= set	
	AX	= error code	

Gets the machine network name under Microsoft Networks.

INT 21h AH = 5E01h — Set Machine Name — ver: 3.1

Input:	CH	= 0, if undefine name
		= nonzero, if define name
	CL	= name number
	DS:ESI	= pointer to 16-byte buffer with ASCIIZ name
Output:	None	

Sets the machine network name under Microsoft Networks.

INT 21h **Set Printer Setup String**
AH = 5E02h **ver: 3.1**

Input:	BX	= redirection list index
	CX	= number of bytes in setup string (<= 64)
	DS:ESI	= pointer to setup string

Output:	If success:	
	Carry flag	= clear
	If failure:	
	Carry flag	= set
	AX	= error code

Specifies a printer setup string under Microsoft Networks.

INT 21h **Get Printer Setup String**
AX = 5E03h **ver: 3.1**

Input:	BX	= redirection list index
	ES:EDI	= pointer to 64-byte buffer

Output:	If success:	
	Carry flag	= clear
	CX	= length of string
	setup string copied into buffer at ES:EDI	
	If failure:	
	Carry flag	= set
	AX	= error code

Obtains a printer setup string under Microsoft Networks.

INT 21h **Get Redirection List Entry**
AX = 5F02h **ver: 3.1**

Input:	BX	= redirection list index
	DS:ESI	= pointer to 16-byte buffer
	ES:EDI	= pointer to 128-byte buffer

Output: If success:
  Carry flag = clear
  BH = device status
  CX = stored parameter value
  ASCIIZ device name stored in buffer at DS:ESI
  ASCIIZ network name stored in buffer at ES:EDI
  DX = destroyed
 If failure:
  Carry flag = set
  AX = error code

Obtains an entry from the redirection list under Microsoft Networks.

INT 21h **Redirect Device**
AX = 5F03h **ver: 3.1**

Input:	BL	= device type
	CX	= stored parameter value
	DS:ESI	= pointer to ASCIIZ device name
	ES:EDI	= pointer to ASCIIZ network name, followed by ASCIIZ password

Output: If success:
  Carry flag = clear
 If failure:
  Carry flag = set
  AX = error code

Redirects a device under Microsoft Networks.

INT 21h		**Cancel Device Redirection**
AX = 5F04h		**ver: 3.1**
Input:	DS:ESI	= pointer to ASCIIZ device name
Output:	If success:	
	Carry flag	= clear
	If failure:	
	Carry flag	= set
	AX	= error code

Cancels a previous device redirection request under Microsoft Networks.

INT 21h		**Resolve Path String**
AH = 60h		**ver: all undoc**
Input:	DS:ESI	= pointer to ASCIIZ relative path string
	ES:EDI	= pointer to 128-byte buffer
Output:	If success:	
	Carry flag	= clear
	AX	= destroyed
	full ASCIIZ path name stored in buffer at ES:EDI	
	If failure:	
	Carry flag	= clear
	AX	= error code

Resolves a relative path string to a canonical path string.

INT 21h **Get Protected-Mode Program Segment Prefix**
AH = 62h **ver: all**

Input:	None	
Output:	BX	= protected mode segment selector of program segment prefix

Returns the protected-mode segment selector (0024h) for the segment mapping the program's PSP. Note that this is not symmetrical with the Set PSP function (50h), which uses the real-mode paragraph address of the PSP. To get the actual conventional memory address of the PSP, use function 51h.

INT 21h **Get Extended Country Information**
AH = 65h **ver: 3.3**

Input:	AL	= subfunction code
	BX	= code page
	CX	= number of bytes in buffer
	DX	= country ID
	ES:EDI	= pointer to buffer

Output: If success:
  Carry flag = clear
  data placed in caller's buffer at ES:EDI
If failure:
  Carry flag = set

Returns information about a country or code page. For subfunctions 2, 4, 6, and 7 the DWORD at offset 01h in the buffer contains an offset in segment 0034h (the MS-DOS memory segment) of a second buffer with additional information. See Reference 3 in the preface for the contents of the buffers.

INT 21h AH = 66h — Get/Set Code Page — ver: 3.3

```
Input:      AL      = 1, if get code page
                    = 2, if set code page
            If AL   = 2:
              BX          = code page to select
```

```
Output:     If success:
              Carry flag    = clear
              If AL         = 1:
                BX              = active code page
                DX              = default code page
            If failure:
              Carry flag    = set
              AX            = error code
```

Gets or sets the current code page.

INT 21h AH = 67h — Set Handle Count — ver: 3.3

```
Input:      BX      = desired number of handles
```

```
Output:     If success:
              Carry flag    = clear
            If failure:
              Carry flag    = set
              AX            = error code
```

Sets the maximum number of file handles allowed for the current program.

INT 21h **Commit File**
AH= 68h **ver: 3.3**

Input:	BX	= file handle

Output:
If success:
  Carry flag = clear
If failure:
  Carry flag = set
  AX = error code

Flushes the file directory entry to disk.

INT 21h **Extended Open/Create File**
AX = 6C00h **ver: 4.0**

Input:	BX	= open mode
	CX	= file attribute
	DX	= action code
	DS:ESI	= pointer to ASCIIZ pathname

Output:
If success:
  Carry flag = clear
  AX = file handle
  CX = 1, if file was opened
     = 2, if file was created
     = 3, if file was replaced
If failure:
  Carry flag = set
  AX = error code

Opens, creates, or replaces a file on disk.

INT 22h — Address of Terminate Routine

ver: all

The real-mode interrupt vector for INT 22h contains the address of the routine that MS-DOS invokes when the currently executing program terminates via a call to INT 21h function 4Ch.

INT 23h — CTRL-C Handler

ver: all

This interrupt is issued in real mode by MS-DOS whenever a DOS function call is made and there is a CTRL-C character in the keyboard input buffer. Programs that wish to be notified when this occurs may take over this interrupt, either to gain control in a real-mode handler (with 386 | DOS-Extender system function 2505h), or to gain control in a protected-mode handler (with 386 | DOS-Extender system function 2506h).

INT 24h — Critical Error Handler

ver: all

This interrupt is issued in real-mode by MS-DOS whenever a critical error occurs. Programs that wish to be notified when this occurs may take over this interrupt, either to gain control in a real-mode handler (with 386 | DOS-Extender system function 2505h), or to gain control in a protected-mode handler (with 386 | DOS-Extender system function 2506h).

INT 25h — Absolute Disk Read

The absolute disk read system call is not supported in protected mode. Programs that need to use this function must use 386 | DOS-Extender system call 2511h, issue real-mode interrupt with segment registers specified, and are responsible for allocating a data buffer that resides in conventional memory.

INT 26h — Absolute Disk Write

The absolute disk write system call is not supported in protected mode. Programs that need to use this function must use 386 | DOS-Extender system call 2511h, issue real-mode interrupt with segment registers specified, and are responsible for allocating a data buffer that resides in conventional memory.

INT 27h — Terminate and Stay Resident

ver: all

Input:	AL	= return code
Output:	None	

The terminate and stay resident function exits back to MS-DOS, while still keeping the program and 386 | DOS-Extender in memory. Any open files remain open. TSR programs must leave as much conventional memory and extended memory as possible free, so that memory is available to run other programs. TSR programs are discussed in more detail in section 7.8.

INT 2Fh AH = 01h — Print Spooler

ver: all

Calls to INT 2Fh function 01h are used to request services from the MS-DOS print spooler.

INT 2Fh **Get Installed Status**
AX = 0100h **ver: all**

Input: None

Output: If success:

Carry flag	= clear	
AL	= status	
	= 00h, if spooler not installed, OK to install it	
	= 01h, if spooler not installed, can't install it	
	= FFh, if spooler is installed	

If failure:

Carry flag	= set
AX	= error code

Returns the installed status of the print spooler. This call must be made to determine whether the print spooler is present before making any other spooler calls.

INT 2Fh **Print File**
AX = 0101h **ver: all**

Input: DS:EDX = pointer to 7-byte packet

Output: If success:

Carry flag	= clear

If failure:

Carry flag	= set
AX	= error code

Submits a file to the spooler to be queued for printing. The packet specifies the level, the file name, and the path of the file to be printed, in the following format:

00h	BYTE	Level
01h	DWORD	Offset of zero-terminated file name string
05h	WORD	Segment selector of file name string

INT 2Fh **Remove File from Queue**
AX = 0102h **ver: all**

Input: DS:EDX = pointer to zero-terminated file name string

Output: If success:
  Carry flag = clear
 If failure:
  Carry flag = set
  AX = error code

The specified file is removed from the print queue. The "*" and "?" wildcard characters are allowed in the file name string.

INT 2Fh **Cancel All Files**
AX = 0103h **ver: all**

Input: None

Output: If success:
  Carry flag = clear
 If failure:
  Carry flag = set
  AX = error code

Cancels all files currently in the print queue.

INT 2Fh **Hold for Status Read**
AX = 0104h **ver: all**

Input:	None

Output:

If success:

Carry flag	= clear
DX	= error count
DS:ESI	= pointer to print queue

If failure:

Carry flag	= set
AX	= error code

Holds jobs in the print queue, so that it can be scanned. Issuing any other spooler call releases the hold. The print queue consists of an array of 64-byte zero-terminated file name strings. The first entry is the file currently being printed. The end of the queue is marked by an entry with a zero in the first byte.

INT 2Fh **End Status Hold**
AX = 0105h **ver: all**

Input:	None

Output:

If success:

Carry flag	= clear

If failure:

Carry flag	= set
AX	= error code

Releases the hold placed on the print queue by function 04h.

INT 2Fh **Check if ASSIGN Utility Installed**
AX = 0600h **ver: 3.2**

Input:	None		
Output:	If success:		
		Carry flag	= clear
		AL	= 0, if not installed, OK to install
			= 1, if not installed, can't install
			= FFh, if installed
	If failure:		
		Carry flag	= clear
		AX	= error code

Checks whether the resident portion of the ASSIGN utility is loaded.

INT 2Fh **Check if SHARE Module Loaded**
AX = 1000h **ver: 3.2**

Input:	None		
Output:	If success:		
		Carry flag	= clear
		AL	= 0, if not installed, OK to install
			= 1, if not installed, can't install
			= FFh, if installed
	If failure:		
		Carry flag	= set
		AX	= error code

Checks whether the SHARE.EXE file sharing module is loaded.

INT 2Fh AX = B700h — Check if APPEND Utility Installed

ver: 3.3

Input: None

Output:

If success:

Carry flag	= clear
AL	= 0, if not installed, OK to install
	= 1, if not installed, can't install
	= FFh, if installed

If failure:

Carry flag	= set
AX	= error code

Checks whether the APPEND utility is installed.

INT 2Fh AX = B704h — Get APPEND Path Pointer

ver: 4.0

Input: None

Output:

If success:

Carry flag	= clear
ES:EDI	= pointer to active APPEND path

If failure:

Carry flag	= set
AX	= error code

Returns a pointer to the active APPEND path string.

INT 2Fh **Get APPEND Function State**
AX = B706h **ver: 4.0**

Input:	None	
Output:	If success:	
	Carry flag	= clear
	BX	= APPEND state
	If failure:	
	Carry flag	= set
	AX	= error code

See Reference 3 for the bit definitions in the state word.

INT 2Fh **Set APPEND Function State**
AX = B707h **ver: 4.0**

Input:	BX	= APPEND state
Output:	If success:	
	Carry flag	= clear
	If failure:	
	Carry flag	= set
	AX	= error code

Please see Reference 3 in the Preface for the bit definitions in the state word.

386 | DOS-Extender System Calls

The 386 | DOS-Extender system calls are made using software interrupt 21h, with a function code in the AX register. These system calls may only be made when executing in protected mode.

The INT 21h interrupt is also used for MS-DOS system calls. A 386 | DOS-Extender system call is signalled by a value in the AH register of 25h. In real mode, this is the set interrupt vector MS-DOS system call. This has been replaced in protected mode by several 386 | DOS-Extender system calls to manipulate interrupt vectors. The function code for the desired 386 | DOS-Extender function is passed in the AL register.

All registers except those used to return results are preserved across 386 | DOS-Extender system calls. All 386 | DOS-Extender system calls return with the processor carry flag set if an error occurred, and with the carry flag clear if the call succeeded. All illegal function codes return with the carry flag set, EAX set to A5A5A5A5h, and all other registers unchanged.

When a 386 | DOS-Extender system call returns failure, an error code is usually placed in the EAX register. Error codes are from the MS-DOS set of error codes (please see Reference 3) plus the following 386 | DOS-Extender-specific error codes:

Error Code	Description
128	LDT buffer too small (Load for Debug, 2512h and 252Ah)
129	invalid parameters passed in to system call
130	not supported under this DPMI host (please see Appendix F for DPMI compatibility)

Error Code	Description
131	interrupt state saving not enabled (Get Interrupt State Calls, 2528h and 2538h)
132	no active interrupt states (Get Interrupt State Calls, 2528h and 2538h)
133	no 386 I VMM page log file (Write Record to Log File, 252Fh)
134	no available 386 I VMM file handles (Map Data File, 252Bh subfunction 3)

Please see the individual call descriptions for error codes returned by a specific system call.

The description of each system call also identifies the 386 I DOS-Extender version in which the function was first supported. Release dates for significant versions of 386 I DOS-Extender include:

1.1	October 17, 1986	first release; first edition of *386 I DOS-Extender Reference Manual*
1.1q	May 27, 1987	first support of 386 I DOS-Extender system calls
1.1w	November 11, 1987	first support of a preliminary (undocumented) version of the VCPI interface
1.1z	February 25, 1988	VCPI version 1.0 supported
2.0	August 1, 1988	second edition of *386 I DOS-Extender Reference Manual* published

2.1c	October 3, 1988	first support for 386 | VMM, the Phar Lap demand-paged virtual memory manager
2.2	February 14, 1988	task switching on hardware interrupts (please see Appendix L)
2.2b	May 30, 1989	workarounds for bugs in the A1 stepping of the 80486 chip
2.2d	October 20, 1989	NEC 9800 series Japanese compatible version of 386 | DOS-Extender introduced
3.0	January 1991	first support of XMS and Windows 3.0 standard mode; removed task switching on hardware interrupts; third edition of *386 | DOS-Extender Reference Manual*

TABLE B-1
386|DOS-EXTENDER SYSTEM CALL SUMMARY

Function Number	Function Name	First Version
2501h	Reset 386 | DOS-Extender Data Structures	1.1q
2502h	Get Protected-Mode Interrupt Vector	1.1q
2503h	Get Real-Mode Interrupt Vector	1.1q
2504h	Set Protected-Mode Interrupt Vector	1.1q
2505h	Set Real-Mode Interrupt Vector	1.1q
2506h	Set Interrupt to Always Gain Control in Protected Mode	1.1q
2507h	Set Real- and Protected-Mode Interrupt Vectors	1.1q
2508h	Get Segment Linear Base Address (see also 2531h)	1.1w

(cont.)

Function Number	Function Name	First Version
2509h[‡]	Convert Linear Address to Physical Address	1.1w
250Ah[†]	Map Physical Memory at End of Segment (see also 252Bh)	1.1w
250Ch	Get Hardware Interrupt Vectors	1.2c
250Dh	Get Real-Mode Link Information	1.1z
250Eh	Call Real-Mode Procedure	1.1z
250Fh[†]	Convert Protected-Mode Address to MS-DOS Address	1.1w
2510h	Call Real-Mode Procedure, Registers Specified	1.2a
2511h	Issue Real-Mode Interrupt, Registers Specified	1.1z
2512h	Load Program for Debugging (see also 252Ah)	1.2c
2513h	Alias Segment Descriptor	1.2a
2514h	Change Segment Attributes (see also 2531h)	1.2a
2515h	Get Segment Attributes (see also 2531h)	1.2c
2516h	Free All Memory Owned by LDT	2.2
2517h	Get Info on DOS Data Buffer (see also 2530h)	2.1c
2518h	Specify Handler for Moved Segments	2.1c
2519h	Get Additional Memory Error Information	2.1c
251Ah[*]	Lock Pages in Memory (see also 252Bh)	2.1c
251Bh[*]	Unlock Pages (see also 252Bh)	2.1c
251Ch[*]	Free Physical Memory Pages (see also 252Bh)	2.1c
		(cont.)

‡ Never available under DPMI.

† Under DPMI, requires specific support: please see Appendix F and the detailed description of this call's functions.

* For use with 386 | VMM.

Function Number	Function Name	First Version
251Dh‡	Read Page Table Entry	2.1c
251Eh‡	Write Page Table Entry	2.1c
251Fh‡	Exchange Two Page Table Entries	2.1c
2520h	Get Memory Statistics	2.2
2521h	Limit Program's Extended Memory Usage (see also 2536h)	2.2
2522h	Specify Alternate Page Fault Handler	2.2
2523h*	Specify Out-of-Swap-Space Handler	2.2
2524h*	Specify Page Replacement Handlers	2.2
2525h	Limit Program's Conventional Memory Usage (see also 2536h)	2.2
2526h	Get Configuration Information	2.2d
2527h	Enable/Disable State Saving on Interrupts	2.2d
2528h	Get Last Protected Mode State after DOS CTRL-C Interrupt	2.2d
2529h	Load Flat Model .EXP or .REX File	2.3
252Ah	Load Program for Debugging	3.0
252Bh	Memory Region Page Management	3.0
252Bh, BH = 0†	Create Unmapped Pages (see also 252Ch)	3.0
252Bh, BH = 1†	Create Allocated Pages	3.0
252Bh, BH = 2†	Create Physical Device Pages (see also 250Ah)	3.0
252Bh, BH = 3*	Map Data File into Allocated Pages	3.0
		(cont.)

‡ Never available under DPMI.

† Under DPMI, requires specific support: please see Appendix F and the detailed description of this call's functions.

* For use with 386 | VMM.

Function Number	Function Name	First Version
252Bh, BH = 4	Get Page Types	3.0
252Bh, BH = 5*	Lock Pages (see also 251Ah)	3.0
252Bh, BH = 6*	Unlock Pages (see also 251Bh)	3.0
252Bh, BH = 7*	Free Physical Memory Pages, Retaining Data (see also 251Ch)	3.0
252Bh, BH = 8*	Free Physical Memory Pages, Discarding Data (see also 251Ch)	3.0
252Ch†	Add Unmapped Pages at End of Segment (see also 252Bh)	3.0
252Dh*	Close 386 | VMM File Handle	2.3
252Eh*	Get/Set 386 | VMM Parameters	2.3
252Fh*	Write Record to 386 | VMM Page Log File	3.0
2530h	Set DOS Data Buffer Size (see also 2517h)	2.3
2531h	Read/Write LDT Segment Descriptor	3.0
2532h	Get Protected-Mode Processor Exception Vector	3.0
2533h	Set Protected-Mode Processor Exception Vector	3.0
2534h	Get Interrupt Flag	3.0
2535h	Read/Write System Registers	3.0
2536h	Minimize/Maximize Extended/Conventional Memory Usage	3.0
2537h	Allocate Conventional Memory Above DOS Data Buffer	3.0
2538h	Get Registers from Interrupt State	3.0
2539h	Get .EXP Header Offset in Bound File	3.0
		(cont.)

‡ Never available under DPMI.

† Under DPMI, requires specific support: please see Appendix F and the detailed description of this call's functions.

* For use with 386 | VMM.

Function Number	Function Name	First Version
253Ah	Install Resize Segment Failure Handler	3.0
253Bh	Jump to Real Mode	3.0
253Ch*	Shrink 386 | VMM Swap File	3.0
25C0h	Allocate MS-DOS Memory Block	1.1w
25C1h	Release MS-DOS Memory Block	1.1w
25C2h	Resize MS-DOS Memory Block	1.1w
25C3h	Execute Program	1.2b

‡ Never available under DPMI.

† Under DPMI, requires specific support: please see Appendix F and the detailed description of this call's functions.

* For use with 386 | VMM.

TABLE B-2
SYSTEM CALLS GR OUPED BY FUNCTIONALITY

Interrupt Control		
2501h	Reset 386 | DOS-Extender Data Structures	1.1q
2502h	Get Protected-Mode Interrupt Vector	1.1q
2503h	Get Real-Mode Interrupt Vector	1.1q
2504h	Set Protected-Mode Interrupt Vector	1.1q
2505h	Set Real-Mode Interrupt Vector	1.1q
2506h	Set Interrupt to Always Gain Control in Protected Mode	1.1q
2507h	Set Real- and Protected-Mode Interrupt Vectors	1.1q
250Ch	Get Hardware Interrupt Vectors	1.2c
2522h	Specify Alternate Page Fault Handler	2.2
2532h	Get Protected-Mode Processor Exception Vector	3.0
2533h	Set Protected-Mode Processor Exception Vector	3.0
2534h	Get Interrupt Flag	3.0
LDT Descriptor Management		
2508h	Get Segment Linear Base Address (see also 2531h)	1.1w
2513h	Alias Segment Descriptor	1.2a
2514h	Change Segment Attributes (see also 2531h)	1.2a
2515h	Get Segment Attributes (see also 2531h)	1.2c
2531h	Read/Write LDT Segment Descriptor	3.0

(cont.)

<u>Memory Management</u>

2509h‡	Convert Linear Address to Physical Address	1.1w
250Ah†	Map Physical Memory at End of Segment (see also 252Bh)	1.1w
2516h	Free All Memory Owned by LDT	2.2
2519h	Get Additional Memory Error Information	2.1c
2520h	Get Memory Statistics	2.2
2521h	Limit Program's Extended Memory Usage (see also 2536h)	2.2
2525h	Limit Program's Conventional Memory Usage (see also 2536h)	2.2
252Bh, BH = 0†	Create Unmapped Pages	3.0
252Bh, BH = 1†	Create Allocated Pages	3.0
252Bh, BH = 2†	Create Physical Device Pages (see also 250Ah)	3.0
252Bh, BH = 4	Get Page Types	3.0
252Ch†	Add Unmapped Pages at End of Segment (see also 252Bh)	3.0
2536h	Minimize/Maximize Extended/Conventional Memory Usage	3.0
48h	DOS Allocate Segment	all
49h	DOS Free Segment	all
4Ah	DOS Resize Segment	all
		(cont.)

‡ Never available under DPMI.

† Under DPMI, requires specific support: please see Appendix F and the detailed description of this call's functions.

* For use with 386 | VMM.

Conventional Memory Management		
2517h	Get Info on DOS Data Buffer (see also 2530h)	2.1c
2525h	Limit Program's Conventional Memory Usage (see also 2536h)	2.2
2530h	Set DOS Data Buffer Size (see also 2517h)	2.3
2536h	Minimize/Maximize Extended/Conventional Memory Usage	3.0
2537h	Allocate Conventional Memory Above DOS Data Buffer	3.0
25C0h	Allocate MS-DOS Memory Block	1.1w
25C1h	Release MS-DOS Memory Block	1.1w
25C2h	Resize MS-DOS Memory Block	1.1w
386 | VMM		
251Ah*	Lock Pages in Memory (see also 252Bh)	2.1c
251Bh*	Unlock Pages (see also 252Bh)	2.1c
251Ch*	Free Physical Memory Pages (see also 252Bh)	2.1c
2522h	Specify Alternate Page Fault Handler	2.2
2523h*	Specify Out-of-Swap-Space Handler	2.2
2524h*	Specify Page Replacement Handlers	2.2
252Bh, BH = 3*	Map Data File into Allocated Pages	3.0
252Bh, BH = 5*	Lock Pages (see also 251Ah)	3.0
252Bh, BH = 6*	Unlock Pages (see also 251Bh)	3.0
252Bh, BH = 7*	Free Physical Memory Pages, Retaining Data (see also 251Ch)	3.0
		(cont.)

‡ Never available under DPMI.

† Under DPMI, requires specific support: please see Appendix F and the detailed description of this call's functions.

* For use with 386 | VMM.

386 | VMM (cont.)		
252Bh, BH = 8*	Free Physical Memory Pages, Discarding Data (see also 251Ch)	3.0
252Dh*	Close 386 | VMM File Handle	2.3
252Eh*	Get/Set 386 | VMM Parameters	2.3
252Fh*	Write Record to 386 | VMM Page Log File	3.0
253Ch*	Shrink 386 | VMM Swap File	3.0
Mixed Real/Protected Mode		
250Dh	Get Real-Mode Link Information	1.1z
250Eh	Call Real-Mode Procedure	1.1z
250Fh†	Convert Protected-Mode Address to MS-DOS Address	1.1w
2510h	Call Real-Mode Procedure, Registers Specified	1.2a
2511h	Issue Real-Mode Interrupt, Registers Specified	1.1z
253Bh	Jump to Real Mode	3.0
Debugging Control		
2512h	Load Program for Debugging (see also 252Ah)	1.2c
2518h	Specify Handler for Moved Segments	2.1c
2527h	Enable/Disable State Saving on Interrupts	2.2d
2528h	Get Last Protected Mode State after DOS CTRL-C Interrupt	2.2d
252Ah	Load Program for Debugging	3.0
2531h	Read/Write LDT Segment Descriptor	3.0
2538h	Get Registers from Interrupt State	3.0
2539h	Get .EXP Header Offset in Bound File	3.0
		(cont.)

† Under DPMI, requires specific support: please see Appendix F and the detailed description of this call's functions.

* For use with 386 | VMM.

Miscellaneous:

251Dh‡	Read Page Table Entry	2.1c
251Eh‡	Write Page Table Entry	2.1c
251Fh‡	Exchange Two Page Table Entries	2.1c
2526h	Get Configuration Information	2.2d
2529h	Load Flat Model .EXP or .REX File	2.3
2535h	Read/Write System Registers	3.0
253Ah	Install Resize Segment Failure Handler	3.0
25C3h	Execute Program	1.2b

‡ Never available under DPMI.

INT 21h AX = 2501h — Reset 386 | DOS-Extender Data Structures — ver: 1.1q

Input:	None
Output:	Carry flag = clear

The 386 | DOS-Extender data structures that are allocated when an interrupt occurs, and when switching between real and protected mode are released. The purpose of this call is to allow a program to restart itself when down one or more levels of mode switching.

Whenever 386 | DOS-Extender switches modes, it allocates a data structure in which to save processor registers. In addition, whenever switching from protected mode to real mode, a stack memory buffer is allocated to be used for the real-mode stack. The data structures are released when the code that gained control after the mode switch terminates, resulting in a switch back to the previous mode.

If an application program simply decides to restart itself without returning through nested levels of mode switches, then the allocated 386 | DOS-Extender data structures will never be freed, with the result that 386 | DOS-Extender may run out of data structures at a later point in the program.

A typical example is a program that takes over the MS-DOS critical error interrupt in real mode; when it gets control in real mode, it may decide to switch to protected mode and restart itself. As part of the restart process, the restart routine should make this system call to release the data structures.

It is very important for the restart routine to set up a known good stack in a segment owned by the application program **before** making this system call. Otherwise, the stack might be in a stack memory buffer allocated by 386 | DOS-Extender on a previous mode switch. Once this system call is made, the stack buffer will be eligible for re-use, which means that a hardware or software interrupt could cause the same stack buffer to be allocated again!

INT 21h **Get Protected-Mode Interrupt Vector**
AX = 2502h **ver: 1.1q**

Input:	CL	= number of interrupt to get
Output:	Carry flag	= clear
	ES:EBX	= address of protected-mode interrupt handler routine

This function is used to save the original value of a protected-mode interrupt vector in the 386 | DOS-Extender shadow IDT (interrupt descriptor table) before modifying it, so that the original vector can be restored at a later time.

For interrupts 08h-0Fh, this function returns the address of the hardware interrupt handler, **not** the processor exception handler. To obtain the processor exception handler address, use system call 2532h. Please see also sections 6.2, 6.3, and 6.8 through 6.8.2.

INT 21h **Get Real-Mode Interrupt Vector**
AX = 2503h **ver: 1.1q**

Input:	CL	= number of interrupt to get
Output:	Carry flag	= clear
	EBX	= address of real-mode interrupt handler routine

The real-mode segment:offset address is returned with the segment paragraph address in the high 16 bits of EBX and the offset in the low 16 bits of EBX. This function is used to save the original value of a real-mode interrupt vector in the real-mode vector table before modifying it, so that the original vector can be restored before the program exits.

For interrupts 08h-0Fh, this function returns the address of the hardware interrupt handler, **not** the processor exception handler. 386 | DOS-Extender always considers real-mode interrupts in the 08h-0Fh range to be hardware interrupts. Please see also sections 6.2, 6.3, and 6.8 through 6.8.2.

INT 21h **Set Protected-Mode Interrupt Vector**
AX = 2504h **ver: 1.1q**

Input:	CL	= number of interrupt to set
	DS:EDX	= address of protected-mode interrupt handler

Output:	Carry flag	= clear

The protected-mode vector for the specified interrupt in the 386 | DOS-Extender shadow IDT is set to point to the specified interrupt handler routine.

For interrupts 08h-0Fh, this function sets the address of the hardware interrupt handler, **not** the processor exception handler. To set the processor exception handler address, use system call 2533h. Please see also sections 6.2, 6.3, and 6.8 through 6.8.2.

INT 21h **Set Real-Mode Interrupt Vector**
AX = 2505h **ver: 1.1q**

Input:	CL	= number of interrupt to set
	EBX	= address of real-mode interrupt handler

Output:	Carry flag	= clear

The vector in the real-mode interrupt table at the beginning of MS-DOS memory is set to point to the specified interrupt handler routine. The real-mode segment:offset address is passed with the segment paragraph address in the high 16 bits of EBX and the offset in the low 16 bits of EBX.

For interrupts 08h-0Fh, this function sets the address of the hardware interrupt handler, **not** the processor exception handler. Please see also sections 6.2, 6.3, and 6.8 through 6.8.2.

INT 21h AX = 2506h — Set Interrupt to Always Gain Control in Protected Mode

ver: 1.1q

Input:	CL	= number of interrupt to set
	DS:EDX	= address of protected-mode interrupt handler
Output:	Carry flag	= clear

This function takes over the specified interrupt such that the protected-mode interrupt handler always gets control, regardless of whether the interrupt occurred in real mode or in protected mode. **This function modifies both the protected-mode interrupt descriptor table and the real-mode interrupt vector table.** Therefore, the original interrupt vectors for both the protected mode and real-mode interrupt should be saved before making this call and restored before the program exits.

This function is useful for taking over hardware interrupts or for setting up a software interrupt to be used for switching from real mode to protected mode.

For interrupts 08h-0Fh, this function installs a handler for the hardware interrupt, **not** the processor exception. To install a handler for a protected-mode processor exception in the 08h-0Fh range, use system call 2533h.

INT 21h AX = 2507h — Set Real- and Protected-Mode Interrupt Vectors

ver: 1.1q

Input:	CL	= number of interrupt to set
	DS:EDX	= address of protected-mode interrupt handler
	EBX	= address of real-mode interrupt handler
Output:	Carry flag	= clear

This function combines the operations performed by system calls 2504h and 2505h. The difference is that this function sets both the real and protected-mode interrupt vectors indivisibly; i.e., interrupts are disabled until both vectors have been modified. This could be important, for example, when setting up separate real and protected-mode interrupt handlers for a hardware interrupt.

For interrupts 08h-0Fh, this function installs a handler for the hardware interrupt, **not** the processor exception. To install a handler for a protected-mode processor exception in the 08h-0Fh range, use system call 2533h.

INT 21h AX = 2508h — Get Segment Linear Base Address

ver: 1.1w

Input:	BX	= segment selector

Output:
- If success:
  - Carry flag = clear
  - ECX = linear base address of segment
- If invalid segment selector:
  - Carry flag = set

The linear base address of the specified segment is extracted from the descriptor entry in the GDT or LDT and returned in ECX. An attempt to read a GDT segment under DPMI always returns an error.

See also Read/Write LDT Segment Descriptor (function 2531h).

INT 21h **Convert Linear Address to Physical Address**
AX = 2509h **ver: 1.1w**

Input:	EBX	= linear address to convert

```
Output:    If success:
             Carry flag   = clear
             ECX          = physical address
           If linear address not mapped in page tables, or if running
             under DPMI:
             Carry flag   = set
             ECX          = destroyed
```

This function converts a linear address to a physical address. The linear address to be converted is obtained by first calling system function 2508h to get the linear base address of a segment, and then adding in the desired offset within the segment. Please see also Read Page Table Entry (function 251Dh).

This system call should **not** be used to obtain the address of real-mode code in a mixed real and protected-mode program. Function 250Fh should be used for that purpose.

Under DPMI, this function always returns failure (carry flag set).

INT 21h AX = 250Ah — Map Physical Memory at End of Segment

ver: 1.1w

Input:	ES	= segment selector in the LDT of segment to modify
	EBX	= physical base address of memory to map; must be a multiple of four kilobytes.
	ECX	= number of physical 4-KB memory pages to map

Output:	If success:	
	Carry flag	= clear
	EAX	= offset in segment of mapped memory
	If failure:	
	Carry flag	= set
	EAX	= error code
		= 8, if insufficient memory (to create page tables), or -NOPAGE used
		= 9, if invalid segment selector
		= 130, if not supported in this DPMI implementation

The limit of the specified segment is increased by the number of pages mapped, and the system page tables are set up to map the specified physical pages. Since the memory is always mapped at the end of the segment, the returned offset of the mapped memory is always equal to the segment limit, plus one, before the system call was made.

This function is generally used to set up a segment to access memory mapped hardware devices. The usual technique is to allocate a separate LDT segment descriptor by calling INT 21h function 48h with a request for zero pages, and then calling this function to map in the desired physical memory. It can also be used to map memory at the end of an existing segment.

Under DPMI, this function always works for mapping physical memory in a zero-sized segment. When mapping at the end of an existing segment, it returns error 130 unless DPMI Device Mapping capability is present. Please see Appendix F.

Note that the free segment (INT 21h function 49h) or reduce memory allocation (INT 21h function 4Ah) system calls are allowed on a segment that has physical memory mapped in by this function.

If a segment that has aliases is modified, the descriptors for all of the aliased segments are automatically updated. Segments 000Ch and 0014h are always aliased. Additional aliases can be created with system call 2513h.

If this call is made for a segment that already has some memory allocated or mapped, the linear base address of the segment in the LDT segment descriptor may change. This side effect is invisible to the application program. A moved segment handler can be installed with function 2518h; the installed handler is called whenever a segment linear base address changes.

Please see also Create Physical Device Pages (function 252Bh subfunction 2).

This call always returns failure if the -NOPAGE switch is used.

INT 21h — Get Hardware Interrupt Vectors

AX = 250Ch — **ver: 1.2c**

Input:	None	
Output:	Carry flag	= clear
	AL	= base interrupt vector for IRQ0 - IRQ7 hardware interrupts
	AH	= base interrupt vector for IRQ8 - IRQ15 hardware interrupts
	BL	= protected-mode interrupt vector for BIOS print screen function

The standard hardware interrupt vectors used under MS-DOS are 08h-0Fh for IRQ0-7, and 70h-77h for IRQ8-15. Normally, this call always returns 08h in AL, and 70h in AH, because 386 | DOS-Extender always presents hardware interrupts on their standard MS-DOS interrupt vectors, even in environments that move the physical hardware interrupts (please see Appendix G).

The **only** circumstance that will cause 386 | DOS-Extender to present hardware interrupts at other than their standard MS-DOS interrupt vectors is when the -HWIVEC switch is used. This switch is provided only for compatibility with previous versions, and its use is strongly discouraged (see Appendix L). If you do not use the -HWIVEC switch, you can **always** assume that hardware interrupts IRQ0-7 are presented on vectors 08h-0Fh, and IRQ8-15 on vectors 70h-77h.

This function also returns the interrupt vector that must be used to invoke the BIOS print screen function from protected mode. By default, 386 | DOS-Extender relocates this system call to interrupt vector 80h. The -PRIVEC command line switch can be used to select a different interrupt. Please see section 6.8.3 for the reasons behind the relocation of this system call.

When executing on a PC or PC/XT with a 386 board, hardware interrupts IRQ8-IRQ15 do not exist. On these systems, a value of zero is always returned in register AH. Use system call 2526h to obtain the machine architecture type.

INT 21h — **Get Real-Mode Link Information**
AX = 250Dh — **ver: 1.1z**

Input:	None	
Output:	Carry flag	= clear
	EAX	= address of real-mode 386 | DOS-Extender procedure that will call through from real mode to a protected-mode routine
	EBX	= real-mode address of intermode call data buffer
	ECX	= size, in bytes, of intermode call data buffer
	ES:EDX	= protected-mode address of intermode call data buffer

The information returned by this system call is used by programs that mix real and protected mode. The real-mode addresses returned in EAX and EBX contain the real-mode segment paragraph address in the high 16 bits and the offset within the segment in the low 16 bits. See Chapter 7 for a discussion on mixing real and protected-mode code.

The intermode call data buffer can vary in size from 0 to 64 kilobytes (the default is zero) and is allocated under the control of the -CALLBUFS command line switch. The buffer is guaranteed to be in conventional memory and is not used by 386 | DOS-Extender in any way — it is entirely for the use of the application program.

Real-mode code in the application program can call protected-mode procedures in the application program by making a FAR call to the 386 | DOS-Extender real-mode routine whose address is returned in EAX. Set up the stack as follows:

```
           Any parameters to be passed to the protected-mode procedure
           DWORD real-mode FAR pointer to parameter block
                 (described below), or zero if none.
SS:SP ->   PWORD protected-mode FAR address of procedure to call
```

and then make a FAR call to the 386 | DOS-Extender routine. When the protected-mode procedure gets control, the stack looks like:

```
           Parameters passed to protected-mode routine
SS:ESP --> QWORD return address from FAR procedure call
```

The protected-mode procedure exits with a FAR return when it has completed its processing.

When the protected-mode procedure gets control, the general registers and the processor flags contain the same values they had when the call to the 386 | DOS-Extender routine was made in real mode. The contents of the segment registers depend on the parameter block passed to the 386 | DOS-Extender routine. If a zero value is passed for the segment part of the parameter block pointer, then DS, ES, FS, and GS are loaded with the same values they had when the protected-mode program originally got control at start-up time. Otherwise, they are loaded with the values in the parameter block, which has the following format:

Offset	Size	Description
00h	WORD	protected-mode DS value
02h	WORD	protected-mode ES value
04h	WORD	protected-mode FS value
06h	WORD	protected-mode GS value

The protected-mode stack is set up to point to the same stack memory that was used for the real-mode stack. A 386 | DOS-Extender segment which maps the first megabyte of memory used by MS-DOS is loaded into the SS register.

When the real-mode procedure gets control again, all general registers and the processor flags contain the same values they contained when the protected-mode procedure executed its return instruction. The stack is set up exactly as it was when the call to the 386 | DOS-Extender routine was made, except that the parameter block pointer and protected-mode procedure pointer on the stack are replaced with zeros. The original real-mode segment register values are restored. If a parameter block was passed to the 386 | DOS-Extender routine, then the segment selectors that were in the segment registers when the protected-mode procedure returned are stored in the parameter block.

INT 21h AX = 250Eh — Call Real-Mode Procedure (ver 1.1z)

Input:	EBX	= address of real-mode procedure to call
	ECX	= number of two-byte WORDs to copy from protected-mode stack to real-mode stack (The protected-mode stack remains unchanged.)

Output: If success:

Carry flag	= clear

All segment registers are unchanged.
All general registers contain whatever values were placed in them by the real-mode procedure
All processor flags except the carry flag are set as they were left by the real-mode procedure

If failure:

Carry flag	= set
EAX	= error code
	= 1, if not enough real-mode stack space

This system call is used to call a real-mode procedure from protected mode. The real-mode procedure address passed in EBX contains the real-mode segment paragraph address in the high 16 bits, and the offset within the segment in the low 16 bits. When the real-mode routine gets control, all general registers and the processor flags contain the same values they had when the system c all was made. The DS segment register is loaded with the same value as CS, and the values of ES, FS, and GS are unspecified. SS:SP are set to point to a stack buffer (which defaults to 1 KB in size and can be increased by the -ISTKSIZE command line switch) allocated by 386 | DOS-Extender. The specified number of parameters are copied to the real-mode stack. The stack is set up as follows when the real-mode routine gets control:

```
          Parameters copied to real-mode stack
SS:SP ->  DWORD real-mode FAR return address
```

When the real-mode procedure completes its processing, it terminates by executing a FAR return. When the protected-mode routine gets control again, all general registers and all processor flags, except the carry flag (which is used to return status from the system call), contain the values that were left in them by the real-mode procedure. The original protected-mode values of all the segment registers are restored.

INT 21h AX = 250Fh — Convert Protected-Mode Address to MS-DOS Address

ver: 1.1w

Input:	ES:EBX	= protected-mode address to convert
	ECX	= length of data, in bytes

Output: If success:
- Carry flag = clear
- ECX = MS-DOS address

If address not mapped in MS-DOS partition, or if not supported in this DPMI implementation:
- Carry flag = set
- ECX = destroyed

This function converts a protected-mode address to a conventional MS-DOS address, if possible. The conventional segment:offset address is returned with the segment paragraph address in the high 16 bits of ECX and the offset in the low 16 bits of ECX.

This function is intended for use with the -REALBREAK switch, to obtain the conventional memory address of code that runs in real mode.

This function returns correct MS-DOS addresses both under MS-DOS and under an EMS emulator supporting VCPI. This call, rather than the call to obtain a physical memory address, should, therefore, always be used to obtain an address for use by real-mode code in a mixed real- and protected-mode program.

This call also can be used to verify that a data buffer or block of code is contiguous in conventional MS-DOS memory, and fits entirely in conventional memory. If the input length in ECX is greater than zero, this call returns an error if the entire specified length is not contiguous or does not reside in conventional memory.

Under DPMI, this function always returns failure (carry flag set) unless DPMI Conventional Memory capability (please see Appendix F) is present.

INT 21h **Call Real-Mode Procedure, Registers Specified**
AX = 2510h **ver: 1.2a**

```
Input:      EBX       = address of real-mode procedure to call
            ECX       = number of 2-byte WORDs to copy from
                          protected-mode stack to real-mode stack
            DS:EDX    = pointer to parameter block
```

```
Output:     If success:
              Carry flag    = clear
              All segment registers are unchanged.
              EDX           = unchanged
              All other general registers contain whatever values
                were placed in them by the real-mode procedure
              All processor flags except the carry flag are set as
                they were left by the real-mode procedure
              Real-mode register values are returned in the
                parameter block
            If failure:
              Carry flag    = set
              EAX           = error code
                            = 1, if not enough real-mode stack space
```

This function is used to call a real-mode procedure from protected mode and is similar to system call 250Eh. The difference is that all the real-mode register values, including segment registers, are specified and returned by this call. The parameter block pointed to by DS:EDX has the following format:

Offset	Size	Description
00h	WORD	Real-mode DS value
02h	WORD	Real-mode ES value
04h	WORD	Real-mode FS value
06h	WORD	Real-mode GS value
08h	DWORD	Real-mode EAX value
0Ch	DWORD	Real-mode EBX value
10h	DWORD	Real-mode ECX value
14h	DWORD	Real-mode EDX value

When the real-mode procedure gets control, the stack is set up as for function 250Eh. The registers given in the parameter block are loaded with their specified values. All other registers and the processor flags contain the same values they had when the system call was made.

When the protected-mode procedure gains control again after the call, all the original protected-mode segment register values are restored, the original EDX (parameter block pointer) is restored, and all the other general registers and all the processor flags, except the carry flag, contain the values left in them by the real-mode procedure. The parameter block contains the real-mode values that were left in the segment registers and EDX by the real-mode procedure, and the entire real-mode processor EFLAGS value (including the carry flag) is stored in the EAX member of the parameter block. The EBX and ECX members in the parameter block are unchanged.

The real-mode procedure is called with a FAR call instruction, and must return with a FAR RET instruction.

INT 21h AX = 2511h — Issue Real-Mode Interrupt, Registers Specified

ver: 1.1z

Input:	DS:EDX	= pointer to parameter block

Output:
All segment registers unchanged
EDX unchanged
All other registers contain the values placed in them by the real-mode interrupt handler
The processor flags are set as they were left by the real mode interrupt handler
Real-mode register values are returned in the parameter block

This system call allows a real-mode interrupt to be issued with all registers, including the segment register values, specified. The parameter block pointed to by DS:EDX has the following format:

Offset	Size	Description
00h	WORD	Interrupt number to issue
02h	WORD	Real-mode DS value
04h	WORD	Real-mode ES value
06h	WORD	Real-mode FS value
08h	WORD	Real-mode GS value
0Ah	DWORD	Real-mode EAX value
0Eh	DWORD	Real-mode EDX value

The interrupt specified in the parameter block is issued in real mode. Before issuing the interrupt, the register values specified in the parameter block are loaded into the appropriate registers. All other registers are unchanged.

When the protected-mode routine gains control again after the system call completes, all the original protected-mode segment register values are restored, the original EDX (parameter block pointer) is restored, and all the other general registers and all the processor flags contain the values left in them by the real-mode interrupt handler. The parameter block contains the real-mode values that were left in the segment registers and EDX by the real-mode interrupt handler. The EAX register contains the value left in it by the real-mode handler; the EAX value in the parameter block is unchanged.

Note that this system call does **not** return status in the carry flag, unlike all other 386 | DOS-Extender system calls. Instead the processor flags are returned exactly as they were left by the real-mode interrupt handler.

INT 21h		**Load Program for Debugging (see also 252Ah)**
AX = 2512h		**ver: 1.2c**

Input:	DS:EDX	= pointer to zero-terminated program name
	ES:EBX	= pointer to parameter block
	ECX	= size, in bytes, of LDT buffer

Output:	If success:	
	Carry flag	= clear
	EAX	= number of segment descriptors in LDT
	If failure:	
	Carry flag	= set
	EAX	= error code
		= 2, if file not found or path invalid
		= 5, if access denied
		= 8, if insufficient memory
		= 10, if environment invalid
		= 11, if invalid file format
		= 128, if LDT too small

This system call, while still supported, is superseded by 252Ah, which adds new options and better error reporting.

For a complete description of what the Load for Debug call does, please see function 252Ah. This call differs from 252Ah in the following details:

- the 2512h call does not support the demand load option under 386|VMM
- the 2512h call does not return a 386|VMM handle for the loaded .EXP file
- the error information returned by the 2512h function is less detailed.

INT 21h — Alias Segment Descriptor
AX = 2513h — ver: 1.2a

Input:

BX	=	segment selector of descriptor (in either LDT or GDT) to make an alias of
CL	=	access rights byte to place in the alias descriptor
CH	=	use type bit (USE16 or USE32) to place in the alias descriptor

Output:

If success:

Carry flag	= clear
AX	= segment selector for created alias

If failure:

Carry flag	= set
EAX	= error code
	= 8, if insufficient memory (LDT is already maximum size, or not enough memory to grow the LDT)
	= 9, if invalid segment selector in BX
	= 130, if under DPMI and this is not a code or data segment

This system call creates an alias in the LDT for the specified segment. The created alias segment is given the access rights byte (byte 5 in the descriptor) specified by the caller. The access rights byte specifies the segment type, as well as the access that is permitted to the segment. See Reference 1 for a more detailed description. The caller also specifies the use type bit for the segment (bit 6 in byte 6 of the descriptor) for the created alias segment. A value of 00h in CH specifies a USE16 segment, and a value of 40h specifies a USE32 segment. All bits other than bit 6 of CH are ignored by this system call.

Note that 386 | DOS-Extender always tracks aliased segments. When the limit or base address of one of a set of aliased segments changes, the segment descriptors for all of the aliased segments are automatically updated.

Segment aliases can be removed by using INT 21h function 49h to free all but one of the aliased segments. When a segment that has aliases is freed, only the segment descriptor is freed; no memory is freed until the last of the aliased segments is freed.

This call always sets the descriptor privilege level (DPL) to the privilege level at which the application runs, regardless of the DPL setting in the CL register. To set an arbitrary DPL value (not recommended), use function 2531h.

Under DPMI, the access rights byte in CL must have bit 4 set to 1, indicating a code or data segment (not a system segment or gate).

INT 21h — Change Segment Attributes
AX = 2514h — **ver: 1.2a**

Input:	BX	= segment selector of descriptor (in either LDT or GDT) to modify
	CL	= new access rights byte to place in the descriptor
	CH	= new use type bit (USE16 or USE32) to place in the descriptor

Output:	If success:	
	Carry flag	= clear
	If failure:	
	Carry flag	= set
	EAX	= error code
		= 9, if invalid selector in BX
		= 130, if under DPMI and not a code or data segment

This system call modifies the access rights byte (byte 5 in the descriptor) and the use type bit (bit 6 in byte 6 of the descriptor) for the specified segment. After modifying the segment descriptor, it reloads all the segment registers to ensure that the new segment attributes are recognized for any segment registers that reference the modified descriptor.

This call always sets the DPL to the privilege level at which the application runs, regardless of the DPL setting in the CL register. To set an arbitrary DPL value (not recommended), use function 2531h.

Under DPMI, the segment must be in the LDT, and the access rights byte in CL must have bit 4 set to 1, indicating a code or data segment (not a system segment or gate).

Please see also Read/Write LDT Segment Descriptor (function 2531h).

INT 21h — AX = 2515h — Get Segment Attributes — ver: 1.2c

Input:	BX	= segment selector of descriptor (in either LDT or GDT) to get attributes of

Output:	If success:	
	Carry flag	= clear
	CL	= access rights byte for segment
	CH	= use type bit (USE16 or USE32) for segment
	ECX<16-31>	= destroyed
	If failure:	
	Carry flag	= set
	EAX	= error code
		= 9, if invalid segment selector in BX

This function retrieves the segment attributes for the specified segment from the appropriate segment descriptor table and returns them to the caller. See Reference 1 for a complete description of segment descriptors.

Under DPMI, the segment must be in the LDT.

Please see also Read/Write LDT Segment Descriptor (function 2531h).

INT 21h AX = 2516h — Free All Memory Owned by LDT

ver: 2.2

Input:	None
Output:	Carry flag = clear

All the descriptors in the LDT are scanned, and the memory allocated to all present code or data descriptors is freed and the descriptor is zeroed. None of the hardwired LDT descriptors except segments 000Ch and 0014h (the program code and data segments) are modified.

This call can only be used when running at level 0 because you must be executing out of the GDT to free all LDT segments. (This call can not be made under DPMI.)

INT 21h AX = 2517h — Get Info on DOS Data Buffer

ver: 2.1c

Input: None

Output:		
	Carry flag	= clear
	ES:EBX	= protected-mode pointer to data buffer
	ECX	= real-mode pointer to data buffer
	EDX	= size of data buffer, in bytes

This call returns pointers to the buffer used to buffer data on DOS or BIOS system calls, and the size of the buffer. The real-mode pointer returned in ECX contains the real-mode segment paragraph address in the high 16 bits, and the offset within the segment in the low 16 bits.

Please see also Set DOS Data Buffer Size (function 2530h).

Note:	The address of the MS-DOS data buffer changes when the Load for Debug (2512h and 252Ah) or the Set DOS Data Buffer Size (function 2530h) calls are made.

INT 21h — AX = 2518h — Specify Handler for Moved Segments

ver: 2.1c

Input:	ES:EBX	= address of handler to be called when a segment is moved
Output:	Carry flag	= clear
	ES:EBX	= address of previous handler

This call specifies a handler routine to be called by 386 | DOS-Extender when the linear base address of a segment changes because the segment size was increased and there were not enough free page table entries immediately following the end of the segment. The purpose of the call is to enable debuggers to update the contents of the 386 breakpoint registers if the base address of a segment with breakpoints or watchpoints set in it changes. The address of the previous handler routine is returned. The default handler installed by 386 | DOS-Extender does nothing.

The handler is invoked with a FAR procedure call, and must terminate with a FAR return instruction when it completes its processing. The handler can modify any general registers, but is required to preserve the segment registers. Two doubleword parameters are passed on the stack:

```
              DWORD      new linear base address of segment
              DWORD      segment selector of segment that moved
SS:ESP -->    QWORD      FAR return address
```

If a segment with aliases gets moved, the handler is called once for each alias.

INT 21h **Get Additional Memory Error Information**
AX = 2519h **ver: 2.1c**

Input:	None.

Output:	Carry flag	= clear
	EAX	= error code
		= 0, if no error
		= 1, if out of physical memory
		= 2, if out of swap space (unable to grow swap file
		= 3, if out of LDT entries and unable to grow LDT
		= 4, if unable to change direct extended memory allocation mark
		= 5, if -MAXPGMMEM switch value exceeded
		= FFFFFFFFh, if no paging is disabled (-NOPAGE switch used)

This call provides additional information when a memory allocator system call returns a memory error (error number 8). The error code returned applies to the last system call made that returned a memory error.

The most likely error code returned when running under 386 | VMM is error 2, out of space on the disk for the swap file. The best way to deal with this situation is to free up some space on the disk, or place the swap file on a different disk (see the *386 | VMM Reference Manual*).

When running without 386 | VMM, the most likely error condition is error 1, out of physical memory. If this error occurs under 386 | VMM, it means the memory allocation request was large enough that 386 | VMM needs to allocate system pages (such as page tables), and there is not enough physical memory available for the required system pages. The probable reason for this is that the application has locked too many pages in memory. Another possibility is an unreasonably large allocation request (e.g., 4 GB) has been made that requires many system pages to be allocated on a machine without much physical memory.

Error 3 means either that the LDT is already its maximum size (64 KB, or 8KB segments), or that there is no physical memory available to grow the LDT.

Error 4 means that no physical memory is available, but that 386 | DOS-Extender expected direct extended memory to be available. Another program (such as a terminate-and-stay-resident program) is attempting to allocate direct extended memory independently. If 386 | DOS-Extender detects this condition, it abandons any attempt to allocate more direct extended memory for fear of conflicts.

Error 5 is returned if the program attempts to increase its total memory allocation beyond the limit specified with the -MAXPGMMEM command line switch.

Error FFFFFFFFh is returned if -NOPAGE is in effect, and the memory error resulted from an attempt to use a system call that is not supported with -NOPAGE.

INT 21h **Lock Pages in Memory**
AX = 251Ah **ver: 2.1c**

This call is used only with 386 | VMM. It locks a specified address range in memory, paging in any virtual pages that were swapped to disk. If this call is used when 386 | VMM is not present, it has no effect. Please see also system call 252Bh subfunction 5.

INT 21h **Unlock Pages**
AX = 251Bh **ver: 2.1c**

This call is used only with 386 | VMM. It unlocks any locked pages in a specified address range. If this call is used when 386 | VMM is not present, it has no effect. Please see also system call 252Bh subfunction 6.

INT 21h **Free Physical Memory Pages**
AX = 251Ch **ver: 2.1c**

This call is used only with 386 | VMM. It optimizes page replacement for applications that track their own memory usage. If this call is used when

386 | VMM is not present, it has no effect. Please see also system call 252Bh subfunctions 7 and 8.

INT 21h **Read Page Table Entry**
AX = 251Dh **ver: 2.1c**

Input:	If BL = 0:	
	ECX	= linear address of page table entry to read
	If BL = 1:	
	ES:ECX	= address of page table entry to read

Output:	If success:	
	Carry flag	= clear
	EAX	= contents of page table entry
	If failure:	
	Carry flag	= set
	EAX	= error code
		= 9, if invalid address or -NOPAGE switch used
		= 130, if running under DPMI

The contents of the page table entry for the specified address are returned to the caller. The address may be anywhere within the address space for which page tables are allocated.

This call is not supported under DPMI.

INT 21h **Write Page Table Entry**
AX = 251Eh **ver: 2.1c**

Input:	EDX	= value to write into page table entry
	If BL	= 0:
	ECX	= linear address of page table entry to write
	If BL	= 1:
	ES:ECX	= address of page table entry to write

Output:	If success:	
	Carry flag	= clear
	If failure:	
	Carry flag	= set
	EAX	= error code
		= 9, if invalid address or -NOPAGE switch used
		= 130, if running under DPMI

The specified value is written to the page table entry for the specified address. The address may be anywhere within the address space for which page tables are allocated. Interrupts are disabled while the page table entry is modified and the translation lookaside buffer (TLB) is flushed. This call can easily crash the system if used incorrectly.

This call is not supported under DPMI.

INT 21h **Exchange Two Page Table Entries**
AX = 251Fh **ver: 2.1c**

Input:	If BL = 0:
	ECX = linear address of first page table entry
	EDX = linear address of second page table entry
	If BL = 1:
	ES:ECX = address of first page table entry
	FS:EDX = address of second page table entry

Output:	If success:		
		Carry flag	= clear
	If failure:		
		Carry flag	= set
		EAX	= error code
			= 9, if invalid address or -NOPAGE switch used
			= 130, if running under DPMI

The values in the page table entries for the two specified addresses are exchanged. The addresses may be anywhere within the address space for which page tables are allocated. Interrupts are disabled while the page table entries are modified and the translation lookaside buffer (TLB) is flushed. This call can easily crash the system if used incorrectly.

This call is not supported under DPMI.

INT 21h **Get Memory Statistics**
AX = 2520h **ver: 2.2**

Input:	DS:EDX	= pointer to buffer at least 100 bytes in size
	BL	= 0, if don't reset 386 | VMM statistics
		= 1, if reset 386 | VMM statistics

Output: Carry flag = clear
100 bytes of buffer at DS:EDX filled in

Statistics about the application's memory usage, and about paging activity if the 386 | VMM subsystem is present, are returned. If BL = 1, then the virtual memory paging statistics are reset after they are read into the buffer. The 100-byte buffer pointed to by DS:EDX is filled in with statistics in the following format:

Offset	Size	Description
00h	DWORD	1, if the 386 | VMM virtual memory subsystem is present; 0, if not present
04h	DWORD	*nconvpg:* the total number of conventional memory pages available, alterable using system call 2525h or 2536h
08h	DWORD	reserved, always zero
0Ch	DWORD	*nextpg:* the total number of extended memory pages from all sources, that is, direct extended memory, VCPI, XMS, and special memory such as COMPAQ built-in memory. It is the sum of memory currently allocated plus what is reported as currently available. (Warning: This must be considered an approximate number if running under a multitasking environment such as DESQview or Windows. It must **not** be relied upon, because in a multitasking environment, another program might allocate it before you can.)

Offset	Size	Description
10h	DWORD	*extlim:* the extended memory page limit (can be set by the application with system call 2521h or 2536h, and defaults to *nextpg*, the total number of available extended memory pages. System call 2521h or 2536h disallows setting *extlim* higher than *nextpg*; but it can become higher after some other program consumes memory, reducing *nextpg*.)
14h	DWORD	*aphyspg*: the total number of physical pages allocated to the application
18h	DWORD	*alockpg*: the total number of locked pages owned by the application
1Ch	DWORD	*sysphyspg*: the total number of physical memory pages allocated to 386 | DOS-Extender for system needs
20h	DWORD	reserved, contents undefined
24h	DWORD	the linear address of the beginning of the address space allocated to the application
28h	DWORD	the linear address of the end of the address space allocated to the application
2Ch	DWORD	the number of seconds since the last time 386 | VMM statistics were reset
30h	DWORD	the total number of page faults since the last time 386 | VMM statistics were reset
34h	DWORD	the number of pages written to the swap file since the last time 386 | VMM statistics were reset

Offset	Size	Description
38h	DWORD	the number of reclaimed pages (page faults on pages that had previously been swapped to disk) since the last time 386 | VMM statistics were reset
3Ch	DWORD	the number of code and data pages allocated to the application (i.e., the size in pages of the linear address space allocated to the application, not including any pages that are unmapped or mapped to a physical device address). 386 | DOS-Extender will not permit this to exceed the size limit specified with -MAXPGMMEM.
40h	DWORD	the size, in pages, of the 386 | VMM swap file
44h	DWORD	reserved, contents undefined
48h	DWORD	the minimum number of conventional memory pages below which the conventional memory block may not be shrunk (program pages placed in conventional memory with the -REALBREAK switch)
4Ch	DWORD	the maximum size, in pages, to which the swap file can be increased (controlled with the -MAXSWFSIZE switch)
50h	DWORD	*flags:* a 32-bit flags doubleword. The only bit currently defined is bit zero, which is set to 1 if a 386 | VMM page fault is in progress (of interest, for example, to a critical error handler). Bits 1-31 are always cleared to zero.
54h	DWORD	*nfreepg*: the number of free physical pages guaranteed available by 386 | DOS-Extender. The total number of physical pages currently consumed by 386 | DOS-Extender on behalf of itself and the application is *aphyspg* + *sysphyspg* + *nfreepg*.

Offset	Size	Description
58h	DWORD	*navailpg*: the number of physical pages currently available, but not guaranteed to remain available in a multitasking environment. This number is always greater than or equal to *nfreepg*. It is equal to MIN(*nextpg, extlim*) - *aphyspg* - *sysphyspg*.

INT 21h **Limit Program's Extended Memory Usage**
AX = 2521h **ver: 2.2**

Input:	EBX	= limit, in pages, of physical extended memory that the application may use

```
Output:     If success:
              Carry flag   = clear
            If failure:
              Carry flag   = set
              EAX          = error code
                           = 8, if insufficient memory or -NOPAGE
                               switch used
              EBX          = maximum limit, in pages
              ECX          = minimum limit, in pages
```

This call sets the limit on the number of pages of physical extended memory which may be used by the application. See also function 2536h, which performs the same operation but may be easier to use. This call is primarily for use with 386 | VMM, but it can be used without 386 | VMM if the program has not allocated all available physical memory.

The current extended memory limit is given by the *extlim* parameter obtained with system call 2520h, Get Memory Statistics. When the program starts up, it may use all the available extended memory on the computer. Under some circumstances, it is desirable to free some physical extended memory for use by another program. The program can continue to run normally after it reduces its usage of physical extended memory; it will just page more frequently because less physical memory is available in which to keep program virtual pages.

Physical memory is commonly freed before performing an EXEC to a child program, so that the child program will have physical memory available for its use. This system call should only be used when EXECing to a protected mode program, since real mode programs do not need to use extended memory. Note that when performing an EXEC to a protected mode child, it is also necessary to make sure that some conventional memory is available to load a second copy of 386 | DOS–Extender. This can be done with system call 2525h, Limit Program's Conventional Memory Usage. After the child program has terminated, these calls are typically used again to raise the limit back to the maximum value, so the application will run as efficiently as possible.

When this system call returns an error, it also returns the minimum and maximum values to which the limit may be set. The maximum value is always equal to the *nextpg* parameter returned by system call 2520h, Get Memory Statistics. The minimum value is usually zero, but may be higher if no physical conventional memory pages are available, or if the application program has locked many virtual pages in memory. This call may return an error if there is insufficient swap space available, since reducing the number of available physical pages causes more virtual pages to be kept on disk.

Using this call may result in considerable reshuffling of physical memory, and in some of the application's virtual memory being paged to disk, if the virtual memory subsystem is present. Locked pages are guaranteed to remain in memory, but they may be moved to different physical pages! Note that, if the application has aliased any locked virtual memory pages with system call 250Ah, Map Physical Memory, the page table entries for the aliased pages are not updated, even though the data may have been moved to a different physical page. Applications which alias virtual memory pages should never make this system call.

If 386 | VMM is not present, the call will only succeed if there are sufficient free pages available, since paging to disk is not an option without virtual memory.

INT 21h **Specify Alternate Page Fault Handler**
AX = 2522h **ver: 2.2**

Input:	ES:EBX	= address of alternate handler to invoke for invalid page faults
Output:	Carry flag	= clear
	ES:EBX	= address of previous alternate page fault handler

Without 386 | VMM, this call is equivalent to using function 2533h to install a handler for the page fault exception. Under 386 | VMM, this call specifies an alternate page fault handler to receive control when a page fault occurs on an unmapped virtual page, or on a page outside the virtual address space allocated to the application. The address of the previous alternate handler is returned. Please see the *386 | VMM Reference Manual* for details.

Programs that install a page fault handler and need to run both with and without 386 | VMM should use this call rather than functions 2532h and 2533h.

INT 21h **Specify Out-of-Swap-Space Handler**
AX = 2523h **ver: 2.2**

This call is used only with 386 | VMM. It installs a handler to be called if 386 | VMM runs out of disk space for the swap file while processing a page fault. If this call is used when 386 | VMM is not present, it has no effect.

INT 21h **Specify Page Replacement Handlers**
AX = 2524h **ver: 2.2**

This call is used only with 386 | VMM. It installs handlers that allow an application to implement its own page replacement policy. If this call is used when 386 | VMM is not present, it has no effect.

INT 21h **Limit Program's Conventional Memory Usage**
AX = 2525h **ver: 2.2**

Input:	EBX	= limit, in pages, of physical conventional memory that the application may use	

Output:	If success:	
	Carry flag	= clear
	If failure:	
	Carry flag	= set
	EAX	= error code
		= 8, if insufficient memory or -NOPAGE switch used
	EBX	= maximum limit, in pages
	ECX	= minimum limit, in pages

System call 2525h sets the limit on the number of pages of physical conventional memory which the application may use. Please see also function 2536h, which performs the same operation but may be easier to use. This call is primarily for use with 386 | VMM, but it can be used without 386 | VMM if the program has not allocated all available physical memory.

The current conventional memory limit is given by the *nconvpg* parameter obtained with system call 2520h, Get Memory Statistics. When the program starts up, by default it is permitted to use all the available conventional memory which is not allocated for MS-DOS or 386 | DOS-Extender. If the -MINREAL and/or -MAXREAL switches are used, some conventional memory is left free at start-up time for other programs, and the protected mode application uses the rest. Under some circumstances, it is desirable to free some physical conventional memory for use by another program. The program can continue to run normally after it reduces its usage of physical conventional memory; it will just page more frequently because less physical memory is available in which to keep program virtual pages.

Physical memory is commonly freed before performing an EXEC to a child program, so that the child program will have physical memory available for its use. It may not be necessary to free up any memory with this system call if the -MINREAL and/or -MAXREAL switches are used, since, in that case, there may already be sufficient free memory to load the child program. Note that some free conventional memory is required even when performing an EXEC to a protected mode child, because a second copy of 386 | DOS-Extender will be loaded in conventional memory. After the child program has terminated, this call is typically used again to raise the limit to the original value, so the application will run as efficiently as possible.

When this system call returns an error, it also returns the minimum and maximum values to which the limit may be set. The maximum value is usually equal to the original limit when the program started up, but may be larger than the original limit if the -MINREAL and/or -MAXREAL switches are used to leave some free memory at load time. The minimum value is usually zero, but may be higher if no physical extended memory pages are available, or if the application program has locked many virtual pages in memory. It will also be nonzero if the application program used the -REALBREAK switch to make sure that part of the program is loaded in contiguous conventional memory. This call may return an error if there is insufficient swap space available, since reducing the number of available physical pages causes more virtual pages to be kept on disk.

Using this call may result in considerable reshuffling of physical memory, and in some of the application's virtual memory being paged to disk, if the virtual memory subsystem is present. Locked pages are guaranteed to remain in memory, but they may be moved to different physical pages! Note that, if the application has aliased any locked virtual memory pages with system call 250Ah, Map Physical Memory, the page table entries for the aliased pages are not updated, even though the data may have been moved to a different physical page. Applications which alias virtual memory pages should never make this system call.

This system call can be made if the virtual memory subsystem is not present, but will only succeed if there are sufficient free pages available, since paging to disk is not an option without virtual memory.

INT 21h AX = 2526h — Get Configuration Information

ver: 2.2d

Input:	DS:EBX	= pointer to 512-byte configuration buffer
	DS:ECX	= pointer to 256-byte buffer for name and path of 386 I VMM swap file

Output:	Carry flag = clear
	Buffers at DS:EBX and DS:ECX filled in

The Get Configuration Information system call returns information about the program environment and the 386 I DOS-Extender command line switch settings. The 256-byte buffer at DS:ECX is filled with the zero-terminated path and name of the 386 I VMM swap file (if 386 I VMM is not present, a null name is returned, i.e., the first byte in the buffer is set to zero). If the swap file name is not needed, point DS:EBX and DS:ECX at the same buffer and only the configuration buffer is returned. The 512-byte buffer at DS:EBX is filled with the following information:

Offset	Size	Description
0000h	DWORD	Flags dword 1
0004h	DWORD	Flags dword 2
0008h	DWORD	Flags dword 3
000Ch	DWORD	386 I DOS-Extender major version number
0010h	DWORD	386 I DOS-Extender minor version number
0014h	DWORD	386 I DOS-Extender letter version (first letter, in ASCII, of the text following the minor version number), 0 if no letter
0018h	DWORD	386 I DOS-Extender beta release flag 0 ==> not a beta release 1 ==> beta release (more than one character following the minor version number)
001Ch	DWORD	Processor ID 3 ==> 386 4 ==> 486

Offset	Size	Description
0020h	DWORD	Coprocessor ID
		4 ==> none
		6 ==> 287
		7 ==> 387 (or 486)
0024h	DWORD	Weitek coprocessor presence
		0 ==> no Weitek coprocessor
		1 ==> Weitek coprocessor present
0028h	DWORD	Machine architecture
		0 ==> IBM compatible
		1 ==> NEC 9800 series compatible
002Ch	DWORD	Machine type
		IBM architecture:
		0 ==> ISA bus (AT-compatible)
		1 ==> MCA bus (PS/2-compatible)
		2 ==> XT bus (PC/XT-compatible with an accelerator board)
		3 ==> EISA bus
		NEC architecture:
		0 ==> normal mode
		1 ==> high resolution mode
0030h	DWORD	VCPI presence flag
		0 ==> VCPI not present
		1 ==> VCPI present
0034h	DWORD	-1167 or -WEITEK switch setting
		0 ==> AUTO (default)
		1 ==> ON
		2 ==> OFF
0038h	DWORD	-MINREAL switch setting (default = 0)
003Ch	DWORD	-MAXREAL switch setting (default = 0)
0040h	DWORD	-MINIBUF switch setting (default = 1)
0044h	DWORD	-MAXIBUF switch setting (default = 64)
0048h	DWORD	size, in bytes, of the data buffer used for DOS system calls
004Ch	DWORD	-NISTACK switch setting (default = 6)
0050h	DWORD	-ISTKSIZE switch setting (default = 1)
0054h	DWORD	-REALBREAK switch setting (default = 0)
0058h	DWORD	-CALLBUFS switch setting (default = 0)

Offset	Size	Description
005Ch	DWORD	-HWIVEC switch setting (default = 0)
0060h	DWORD	-PRIVEC switch setting (default = 80h)
0064h	DWORD	-INTMAP switch setting (default = 0)
0068h	DWORD	reserved, always zero
006Ch	DWORD	interrupt vector used for IRQ0
0070h	DWORD	interrupt vector used for IRQ8 (0 if running on a PC/XT which has only IRQ0-7)
0074h	DWORD	protected-mode interrupt vector used for BIOS print screen system call (0 if not available)
0078h	DWORD	-EXTLOW switch setting (default = 1 MB)
007Ch	DWORD	-EXTHIGH switch setting (default = FFFFF000h)
0080h	DWORD	physical address of lowest direct extended memory page 386 | DOS-Extender can allocate
0084h	DWORD	physical address of 1 byte past the end of the highest direct extended memory page 386 | DOS-Extender can allocate
0088h	DWORD	physical base address of "special" memory, such as Compaq built-in memory, which has been allocated by 386 | DOS-Extender (0 if none)
008Ch	DWORD	size, in bytes, of “special” memory allocated by 386 | DOS-Extender (0 if none)
0090h	DWORD	-MAXVCPIMEM switch setting (default = FFFFFFFFh)
0094h	DWORD	-VSCAN switch setting (default = 4000)
0098h	DWORD	-SWAPCHK switch setting 0 ==> OFF 1 ==> ON 2 ==> FORCE (default) 3 ==> MAX
009Ch	DWORD	-CODESIZE switch setting (default = 0)
00A0h	DWORD	-MINSWFSIZE switch setting (default = 0)
00A4h	DWORD	-MAXSWFSIZE switch setting (default = FFFFFFFFh)
00A8h	DWORD	386 | VMM page replacement policy 0 ==> Least-Frequently-Used (LFU) 1 ==> Not-Used-Recently (NUR)
00ACh	DWORD	-NGDTENT switch setting (default = 0)
00B0h	DWORD	-NLDTENT switch setting (default = 0)
00B4h	DWORD	current privilege level (0-3)

Values from offset 00B8h on were added in version 3.0 of 386 | DOS-Extender.

Offset	Size	Description
00B8h	DWORD	-LOCKSTACK switch setting (default = size of initial stack segment, set automatically by linker)
00BCh	DWORD	-MAXEXTMEM switch setting (default = FFFFFFFFh)
00C0h	DWORD	-MAXXMSMEM switch setting (default = FFFFFFFFh)
00C4h	DWORD	-MAXPGMMEM switch setting (default = FFFFFFFFh)
00C8h	DWORD	-DATATHRESHOLD switch setting (default = 1024)
00CCh	DWORD	386 | VMM presence flag 0 ==> 386 | VMM not present 1 ==> 386 | VMM present
00D0h	DWORD	Cyrix EMC87 coprocessor presence 0 ==> no Cyrix coprocessor 1 ==> Cyrix coprocessor present
00D4h	DWORD	-CYRIX switch setting 0 ==> AUTO (default) 1 ==> ON 2 ==> OFF
00D8h	DWORD	DPMI presence flag 0 ==> DPMI not present 1 ==> DPMI present
00DCh	DWORD	DPMI major version number
00E0h	DWORD	DPMI minor version number
00E4h	DWORD	DPMI capabilities flags (please see Appendix F): 00000001h set if Paging Support capability present 00000002h set if Device Mapping capability 00000004h set if Conventional Memory capability 00000008h set if Exceptions Restartability capability 00000010h set if Page Accessed/Dirty capability FFFFFFE0h reserved, always zero
00E8h	DWORD	VCPI major version number
00ECh	DWORD	VCPI minor version number
00F0h	WORD	Under VCPI, actual base vector for hardware interrupts IRQ0-IRQ7

Offset	Size	Description
00F2h	WORD	Under VCPI, actual base vector for hardware interrupts IRQ8-IRQ15
00F4h	DWORD	XMS presence flag 0 ==> XMS not present 1 ==> XMS present
00F8h	DWORD	XMS major version number
00FCh	DWORD	XMS minor version number
0100h	WORD	program's code segment selector
0102h	WORD	program's data segment selector
0104h	WORD	program's PSP segment selector
0106h	WORD	program's environment segment selector
0108h	WORD	segment selector for segment mapping the 1 MB MS-DOS memory space
010Ah	WORD	segment selector for segment mapping the video text memory
010Ch	WORD	segment selector for segment mapping the video graphics memory on NEC machines (on IBM machines, this segment maps the same memory as the video text segment)
010Eh	WORD	segment selector for segment mapping the Weitek memory space (0 if Weitek chip not present)
0110h	WORD	segment selector for segment mapping the Cyrix memory space (0 if Cyrix chip not present)
0112h	WORD	reserved, set to zero
0114h	DWORD	real mode FAR addr of 386 | DOS-Extender routine to call to jump to protected mode with no saved context (please see system call 253Bh)
0118h	DWORD	current size, in bytes, of LDT
011Ch	228 BYTES	set to zero, reserved for future expansion

Flags DWORD 2 and 3 are currently unused and are always set to zero. Bit definitions for flags DWORD 1 are:

Bit mask	Description
00000001h	set if -NOPAGE switch used
00000002h	set if -A20 switch used
00000004h	set if -VDISK switch used
00000008h	set if -XT switch used
00000010h	set if -AT switch used
00000020h	set if -MCA switch used
00000040h	set if -EISA switch used
00000080h	set if -NORMRES switch (for NEC) used
00000100h	set if -HIGHRES switch (for NEC) used
00000200h	set if -SWFGROW1ST option (the default) selected, cleared if -NOSWFGROW1ST specified
00000400h	set if -NOVM switch used
00000800h	set if -SAVEREGS switch used
00001000h	set if -FWAIT switch used
00002000h	set if -NOVCPI switch used
00004000h	set if -NOMUL switch used
00008000h	set if -NOBMCHK switch used
00010000h	set if -NOSPCLMEM or -NOBIM switch used
00020000h	set if -NOPGEXP switch used
00040000h	set if -SWAPDEFDISK switch used
00080000h	set if -SAVEINTS switch used
00100000h	set if -NOLOAD switch used
00200000h	set if -PAGELOG switch used
00400000h	set if -OPENDENY switch used
00800000h	set if -ERRATA17 switch used
FF000000h	unused, always cleared to zero

INT 21h AX = 2527h — Enable/Disable State Saving on Interrupts

ver: 2.2d

Input:	EBX	= 0, if disable interrupt state saving
		= 1, if enable interrupt state saving
Output:	Carry flag	= clear
	EBX	= previous interrupt state saving flag

This call is used to enable or disable saving the entire register set (the "interrupt state") whenever an interrupt occurs in protected mode. If interrupt state saving is enabled, INT 21h functions 2528h and 2538h can be used to walk through active interrupt states to obtain information about active interrupts. Enabling interrupt state saving increases the overhead of the 386 | DOS-Extender interrupt processing approximately 12%.

Interrupt state saving is intended for use by programs such as debuggers and code profilers.

INT 21h AX = 2528h — Get Last Protected Mode State After DOS CTRL-C Interrupt

ver: 2.2d

Input: DS:EBX = pointer to 60-byte buffer

Output:

If success:
- Carry flag = clear
- Buffer at DS:EBX filled in

If failure:
- Carry flag = set
- EAX = 131, if interrupt state saving not enabled
- = 132, if no active protected mode software interrupt

This call is used to obtain a protected mode program state of interest after getting a MS-DOS CTRL-C interrupt in real mode. It can only be used if interrupt state saving is enabled (INT 21h function 2527h). It is intended for use by a debugger, for getting the program state of most interest to the programmer after a CTRL-C is typed.

This call walks back through active protected mode interrupt states looking for the first protected mode interrupt state that is **not** a hardware interrupt, or INT 21h function 250Eh, 2510h, or 2511h. The rationale is if the application used functions 250Eh, 2510h, or 2511h to call real mode code, the application is executing in real mode and protected mode states are not of interest. If this heuristic does not seem correct, use INT 21h function 2538h to walk the active interrupt states directly.

The registers at the time the interrupt occurred are returned in the buffer at DS:EBX in the following format:

Offset	Size	Description
0000h	8 BYTEs	unused, not modified by this call
0008h	DWORD	EAX value
000Ch	DWORD	EBX value
0010h	DWORD	ECX value
0014h	DWORD	EDX value
0018h	DWORD	ESI value
001Ch	DWORD	EDI value
0020h	DWORD	EBP value
0024h	DWORD	ESP value
0028h	WORD	CS value
002Ah	WORD	DS value
002Ch	WORD	SS value
002Eh	WORD	ES value
0030h	WORD	FS value
0032h	WORD	GS value
0034h	DWORD	EIP value
0038h	DWORD	EFLAGS value

INT 21h **Load Flat Model .EXP or .REX File**
AX = 2529h **ver: 2.3**

Input:	DS:EDX	= pointer to zero-terminated program name string
	ES:EBX	= pointer to 64-byte parameter block
	ECX	= 0, if read program into memory immediately
		= 1, if let program get demand-paged in from .EXP file

Output: If success:
- Carry flag = clear
- EAX = 386 | VMM handle, -1 if no handle
- parameter block loaded with initial register values

If failure:
- Carry flag = set
- EAX = 2, if file error
  = 8, if insufficient memory
- If EAX = 8 (memory error)
  - EBX = 1, if out of physical memory
    = 2, if out of swap space
    = 3, if unable to grow LDT
    = 4, if unable to change extended memory allocation mark
    = 5, if insufficient virtual memory space (value specified by -MAXPGMMEM exceeded)
    = 6, if insufficient conventional memory to satisfy -REALBREAK switch
- If EAX = 2 (file error)
  - EBX = 1, if MS-DOS error opening file
    = 2, if MS-DOS error seeking in file
    = 3, if MS-DOS error reading from file
    = 4, if not a flat model .EXP or .REX file
    = 5, if invalid file format
    = 6, if program linked with a -OFFSET value which is not a multiple of 4K

		= 7, if running with -NOPAGE and program linked with -REALBREAK or -OFFSET
If EBX		= 1, 2, or 3 (MS-DOS error)
	ECX	= error code returned by MS-DOS, including (but not necessarily limited to):
		= 2, if file not found
		= 3, if path not found
		= 4, if too many open files
		= 5, if access denied
		= 6, if invalid file handle
		= 12, if file open access code invalid
		= 32, if sharing violation

The parameter block has the following format:

Offset	Size	Description
00h	DWORD	Initial EIP value
04h	WORD	Initial CS value
O6h	DWORD	Initial ESP value
0Ah	WORD	Initial SS value
0Ch	WORD	Initial DS value
0Eh	WORD	Initial ES value
10h	WORD	Initial FS value
12h	WORD	Initial GS value
14h	DWORD	minimum size of program segment, in bytes (the value that's stored at offset 5Ch in the PSP for the main program).
18h	DWORD	total amount of extra bytes of memory allocated for the program (the value stored at offset 64h in the PSP for the main program).
1Ch-3Fh		reserved, always zeroed

The specified program is loaded into memory, and a code segment and a data segment are created in the LDT for the program. No new PSP or environment are created for the loaded program.

The caller can specify that the program not be initially read into memory, and be paged in as it is referenced. If the program is a packed .EXP file, or if 386 | VMM is not being used, this parameter is ignored and the program is always read into memory.

The created code and data segments are automatically registered as segment aliases, just as the initial application code and data segments (0Ch and 14h) are. This means that whenever the limit or base address for one of the segments changes, the other segment is also automatically updated. Aliased segments can also be created with the Alias Segment Descriptor system call (function 2513h).

The .EXP file is opened for read access only, no child inheritance of the file handle, and with MS-DOS sharing mode set to compatibility mode. If a 386 | VMM handle is returned, the file is kept open so that 386 | VMM can page out of the .EXP file to minimize load times and the space requirements for the swap file. The 386 | VMM handle is not the same as the MS-DOS file handle, and cannot be used by the application to perform file I/O. The file is automatically closed when the application terminates.

When the application has finished using the loaded program, it can free up all the resources allocated to the loaded program by:

1. Freeing the program's code and data segments (both must be freed, since they are aliases) and any other segments allocated by the program after it was loaded, with the Free Segment system call (INT 21h function 49h).

2. Closing the .EXP file, if a 386|VMM handle was returned, with the Close 386|VMM Handle system call (INT 21h function 252Dh).

INT 21h **Load Program for Debugging**
AX = 252Ah **ver: 3.0**

Input:	DS:EDX	= pointer to zero-terminated program name string
	ES:EBX	= pointer to parameter block
	ECX	= size, in bytes, of LDT buffer
	ESI	= 0, if read program into memory immediately
		= 1, if let program get demand-paged in from .EXP file

Output: If success:

Carry flag	= clear	
EAX	= 386 | VMM handle, -1 if no handle	
ECX	= number of segment descriptors in LDT buffer	

If failure:

Carry flag	= set	
EAX	= 2, if file error	
	= 8, if insufficient memory	
	= 128, if LDT buffer too small	
If EAX	= 8 (memory error)	
EBX		= 1, if out of physical memory
		= 2, if out of swap space
		= 4, if unable to change extended memory allocation mark
		= 5, if insufficient virtual memory space (value specified by -MAXPGMMEM exceeded)
		= 6, if insufficient conventional memory to satisfy -REALBREAK switch
		= 7, if unable to allocate conventional memory for program's PSP and environment
If EAX	= 2 (file error)	
EBX		= 1, if MS-DOS error opening file
		= 2, if MS-DOS error seeking in file
		= 3, if MS-DOS error reading from file

```
                   = 4, if not a flat model .EXP or .REX file
                   = 5, if invalid file format
                   = 6, if program linked with a -OFFSET
                   value which is not a multiple of 4K
                   = 7, if running with -NOPAGE and
                      program linked with -REALBREAK or
                      -OFFSET
                   = 8, if MS-DOS error loading real-mode
                      .EXE file
If EBX             = 1, 2, 3, or 8 (MS-DOS error occurred)
   ECX                = error code returned by MS-DOS,
                         including (but not necessarily
                         limited to):
                      = 2, if file not found
                      = 3, if path not found
                      = 4, if too many open files
                      = 5, if access denied
                      = 6, if invalid file handle
                      = 8, if insufficient memory to load
                         real-mode program
                      = 10, if environment invalid when
                         loading real-mode program
                      = 11, if file format of real-mode .EXE
                         file invalid
                      = 12, if file open access code invalid
                      = 32, if sharing violation
```

This call should be used in preference to function 2512h, which returns less detailed error information and does not return a 386 | VMM file handle.

The specified program is loaded for debugging, and a separate PSP and MS-DOS environment are created for the loaded program. The caller can specify that the program not be initially read into memory, and be paged in as it is referenced. If the program is a packed .EXP file, or if 386 | VMM is not used, this parameter is ignored and the program is always read into memory.

This system call is intended for use by debuggers and should not be used if all that is required is an EXEC to another program. It loads a program into memory and initializes the segment descriptors in the caller's LDT buffer; the caller is responsible for copying the descriptors to the actual LDT, as described below. The parameter block pointed to by ES:EBX has the following format:

Inputs:

Offset	Size	Description
00h	DWORD	Offset of environment string
04h	WORD	Segment of environment string
06h	DWORD	Offset of command tail string
0Ah	WORD	Segment of command tail string
0Ch	DWORD	Offset of LDT buffer (size specified in ECX)
10h	WORD	Segment of LDT buffer

Outputs:

Offset	Size	Description
12h	WORD	Real-mode paragraph address of PSP for loaded program
14h	WORD	Real-protected mode flag (0 = real mode, 1 = protected mode)
16h	DWORD	Initial EIP value
1Ah	WORD	Initial CS value
1Ch	DWORD	Initial ESP value
20h	WORD	Initial SS value
22h	WORD	Initial DS value
24h	WORD	Initial ES value
26h	WORD	Initial FS value
28h	WORD	Initial GS value

If the environment string segment selector is zero, the loaded program inherits the parent's environment. Otherwise, it is a sequence of zero-terminated strings, with the whole environment string terminated by an additional zero byte. The command tail string must be present, and the first byte gives the length of the string, not including the count byte or the terminating carriage return. The count byte is followed by the string, which is terminated by a carriage return.

The caller's LDT buffer must be at least as large as the size passed in ECX. On return, as many segment descriptors as required for the loaded program are initialized in the LDT buffer. A minimum of eight descriptors are initialized (please see Table 3-1). Any uninitialized descriptors in the LDT buffer are zeroed. All the fields from offset 12h up in the parameter block are filled in with the program's initial state when the system call returns.

After the program is loaded, the current PSP setting in MS-DOS is still the parent's (debug program's) PSP. The CTRL-C, critical error, and terminate vectors in the child's PSP are not modified, so they still point to MS-DOS. The MS-DOS DTA buffer is set to point into the child's PSP, but since 386 | DOS-Extender always sets the MS-DOS DTA pointer to its own internal buffer on any protected-mode system calls that use the DTA, this should have no effect on protected-mode programs.

This call can be used to load both protected-mode and real-mode programs, and can be used to load protected-mode programs that have 386 | DOS-Extender bound in. To avoid confusing differences in program behavior, always make sure the debugger is run under the same version of 386 | DOS-Extender used for the child program when it is run without the debugger.

Note that this call will fail if there is not enough free memory to load the program. The debugger should either be linked with the -MAXDATA switch to limit its memory usage, or should free up memory at run time. When debugging real-mode programs, conventional memory must be left free by running the debugger with the -MINREAL and/or -MAXREAL switches, or by using the 2525h or 2536h system calls.

After making this call, the debugger needs to copy the child program's segment descriptors from the LDT buffer to the actual LDT. The debugger should first move its own segments elsewhere in the LDT to make room for the child's segments. Additional LDT descriptors can be allocated by requesting a zero-sized segment with the Allocate Segment system call (function 48h). Segment descriptors can be read and written with system call 2531h.

When writing the child program's descriptors to the LDT, the debugger must register segments 000Ch and 0014h in the child as aliased segments, using system call 2513h.

The debugger should use function 50h to set the current PSP to the child program's PSP before transferring control to the child program, and should set the current PSP back to the debugger's PSP after regaining control from the child.

To kill a child program, the debugger should force it to execute a DOS terminate call, so MS-DOS can free up the child's PSP and environment segments and close any open files. When the debugger regains control again after the child terminates, it should free up memory allocated to the child's code and data segments with system call 49h.

INT 21h AX = 252Bh — Memory Region Page Management

ver: 3.0

```
Input:        BH      = subfunction code
                      = 0, to create unmapped pages
                      = 1, to create allocated pages
                      = 2, to create physical device pages
                      = 3, to map data file into allocated pages
                      = 4, to get page types
                      = 5, to lock pages
                      = 6, to unlock pages
                      = 7, to free physical memory pages, retaining
                           data
                      = 8, to free physical memory pages, discarding
                           data
              If BL   = 0
                ECX       = linear address of start of region
              If BL   = 1
                ES:ECX    = address of start of region
              EDX     = length of region, in bytes
              [For other inputs, please see description of specific
                subfunctions.]
```

Output: If success:
  Carry flag = clear
  [For other Outputs, please see description of specific subfunction]
 If failure:
  Carry flag = set
  [For error values, please see description of specific subfunction]

The Memory Region Page Management call is used to modify a range of pages within any segment owned by the application program. Pages can be in one of three states:

Allocated pages are normal memory pages which contain application code and data. If the program does not use 386 | VMM, allocated pages are always physical RAM memory pages. Under 386 | VMM, allocated pages may be paged out to the swap file, and are loaded into memory when the application program references the page. Allocated pages are created when the application program is loaded, or when it allocates a segment or increases the size of a segment, or when it calls subfunction 1 of this system call.

Physical Device pages are used to access memory-mapped devices anywhere in the 4 GB physical address space of the processor. Physical device pages do not consume any physical RAM memory or 386 | VMM swap file resources. Physical device pages are created when the application program calls subfunction 2 of this system call, or calls function 250Ah (Map Physical Memory).

Unmapped pages are pages which are marked not present and which unconditionally cause a page fault when referenced by the application (regardless of whether 386 | VMM is present). By default, if a page fault occurs on an unmapped page 386 | DOS-Extender will terminate the application. The application can install its own page fault handler for page faults on unmapped pages with function 2522h, Specify Alternate Page Fault Handler. Unmapped pages are created at load time at the beginning of the application's code and data segment if the program is linked with the -OFFSET switch. The -OFFSET switch is normally used for detection of null pointer references by the application program. The application can use subfunction 0 of this system call to create unmapped pages anywhere within a segment, and can use system call 252Ch to add unmapped pages to the end of a segment. If the application then installs an alternate page fault handler, it can, for example, use unmapped page references to dynamically grow a data structure as needed.

The **application memory region** is defined by the starting address and length passed in to this call. Because the functionality provided by this call relies on the paging hardware of the processor, the requested action is always performed on units of 4 KB pages; i.e., the **page-aligned memory region** on which the action is performed must always begin and end on a 4 KB address boundary. For some subfunctions, the application memory region can begin and end on any byte boundary; other subfunctions require the caller to page-align the application memory region.

The following rules are applied when converting the application memory region to a page-aligned memory region:

- ☛ For all subfunctions, it is important that the requested action be performed on the entire application memory region. Therefore, partial pages at the beginning and end of the application memory region are always included in the page-aligned memory region.
- ☛ For some subfunctions, the requested action causes loss of data, and therefore should not be permitted to affect any memory outside the application memory region. For these subfunctions, the application memory region is required to be page-aligned, and error 9 (invalid memory region) is returned if the region does not start on a 4 KB boundary or its length is not a multiple of 4 KB.

INT 21h
AX = 252Bh
BH = 0

Create Unmapped Pages
ver: 3.0

Input:	If BL = 0
	ECX = linear address of start of region
	If BL = 1
	ES:ECX = address of start of region
	EDX = length of region, in bytes

Output:	If success:
	Carry flag = clear
	If failure:
	Carry flag = set
	EAX = 8, if memory error
	= 9, if invalid memory region
	= 130, if not supported under this DPMI implementation

The Create Unmapped Pages call is used to change allocated or physical device pages to unmapped pages. If any of the pages were previously allocated, the memory is freed and their contents are discarded. The application memory region must be page-aligned.

Under DPMI, this subfunction returns error 130 unless DPMI Paging Support is present. Please see Appendix F.

INT 21h
AX = 252Bh
BH = 1

Create Allocated Pages
ver: 3.0

```
Input:      If BL    = 0
              ECX      = linear address of start of region
            If BL    = 1
              ES:ECX  = address of start of region
            EDX    = length of region, in bytes
```

```
Output:     If success:
              Carry flag    = clear
            If failure:
              Carry flag    = set
              EAX           = 8, if memory error
                            = 9, if invalid memory region
                            = 130, if not supported under this DPMI
                                 implementation
```

The Create Allocated Pages call is used to change unmapped or physical device pages to allocated pages. Subfunction 0 (create unmapped pages) is first performed on the memory region to ensure that the pages are unmapped. All the pages in the region are then converted to allocated pages, with their contents initialized to zero. The application memory region must be page-aligned.

Under DPMI, this subfunction returns error 130 unless DPMI Paging Support is present. Please see Appendix F.

INT 21h
AX = 252Bh
BH = 2

Create Physical Device Pages

ver: 3.0

Input:	If BL = 0	
	ECX	= linear address of start of region
	If BL = 1	
	ES:ECX	= address of start of region
	EDX	= length of region, in bytes
	EDI	= page-aligned physical base address of device

Output:	If success:	
	Carry flag	= clear
	If failure:	
	Carry flag	= set
	EAX	= 8, if memory error
		= 9, if invalid memory region
		= 130, if not supported under this DPMI implementation

The Create Physical Device Pages call is used to change unmapped or allocated pages to physical device pages. Subfunction 0 (create unmapped pages) is first performed on the memory region to ensure that the pages are unmapped. The specified page-aligned physical memory region is then mapped in the application memory region, with page-level caching disabled if the processor is a 486. The application memory region must be page-aligned.

Under DPMI, this subfunction returns error 130 unless the DPMI Device Mapping capability is present. Please see Appendix F.

INT 21h **Map Data File into Allocated Pages**
AX = 252Bh **ver: 3.0**
BH = 3

This call is used only with 386 | VMM. It maps disk files into a virtual memory region so the file is automatically paged into memory when referenced. If this call is used when 386 | VMM is not present, it has no effect.

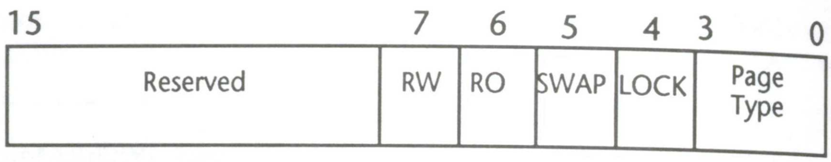
INT 21h **Get Page Types**
AX = 252Bh **ver: 3.0**
BH = 4

Inputs:	If BL	= 0	
	ECX		= linear address of start of region
	If BL	= 1	
	ES:ECX		= address of start of region
	EDX	= length of region, in bytes	
	DS:EDI	= pointer to buffer, one WORD for each 4 KB page in the memory region	

Outputs:	If success:	
	Carry flag	= clear
	If failure:	
	Carry flag	= set
	EAX	= 9, if invalid memory region

The Get Page Types call is used to obtain the page type (allocated, physical device, or unmapped) for each of the pages in the application memory region. The page types are returned as one WORD per 4 KB page in the buffer at DS:EDI. The beginning of the application memory region is required to be page-aligned; there is no alignment requirement on the length.

The WORD of information returned for each page in the region contains the following data:



Bit Mask	Description
000Fh	Page type code: 0, if unmapped page 1, if allocated page 2, if physical device page 3-15, reserved
0010h	Set if page is locked, cleared if unlocked
0020h	Set if page currently swapped to disk, cleared if in physical memory
0040h	Set if page is mapped to a read-only data file
0080h	Set if page is mapped to a read/write data file
FF00h	Reserved, always cleared.

INT 21h
AX = 252Bh
BH = 5

Lock Pages
ver: 3.0

This call is used only with 386 | VMM. It locks a specified address range in memory, paging in any virtual pages that were swapped to disk. If this call is used when 386 | VMM is not present, it has no effect. Please see also system call 251Ah.

INT 21h **Unlock Pages**
AX = 252Bh **ver: 3.0**
BH = 6

This call is used only with 386 | VMM. It unlocks any locked pages in a specified address range. If this call is used when 386 | VMM is not present, it has no effect. Please see also system call 251Bh.

INT 21h **Free Physical Memory Pages, Retaining Data**
AX = 252Bh **ver: 3.0**
BH = 7

This call is used only with 386 | VMM. It optimizes page replacement for applications that track their own memory usage. If this call is used when 386 | VMM is not present, it has no effect. Please see also system call 251Ch.

INT 21h **Free Physical Memory Pages, Discarding Data**
AX = 252Bh **ver: 3.0**
BH = 8

This call is used only with 386 | VMM. It optimizes page replacement for applications that track their own memory usage. If this call is used when 386 | VMM is not present, it has no effect. Please see also system call 251Ch.

INT 21h AX = 252Ch — Add Unmapped Pages at End of Segment

ver: 3.0

Input:	BX	= segment selector
	ECX	= number of 4K pages to add to segment

Output:

If success:

Carry flag	= clear
EAX	= offset in segment of start of unmapped pages

If failure:

Carry flag	= set
EAX	= 8, if insufficient memory
	= 9, if invalid segment selector
	= 130, if not supported under this DPMI implementation

This call increases the size of the segment by the specified number of pages, creating unmapped pages. The returned offset of the start of the unmapped pages is equal to the segment limit, plus one, before the call was made.

The other two calls that can be used to increase the size of a segment are INT 21h function 4Ah (Resize Segment), which adds allocated pages at the end of a segment, and INT 21h function 250Ah (Map Physical Memory), which adds physical device pages at the end of a segment.

The linear base address of the segment may change when this call is made. The side effect is invisible to the application program. A moved segment handler can be installed with function 2518h; the installed handler is called whenever a segment linear base address changes.

Under DPMI, this function returns error 130 unless DPMI Paging Support is present. Please see Appendix F.

INT 21h **Close 386 | VMM File Handle**
AX = 252Dh **ver: 2.3**

This call is used only with 386 | VMM. It is used to close an .EXP or data file kept open by 386 | VMM to swap unmodified pages, reducing space requirements for the swap file. If this call is made when 386 | VMM is not present, it always returns success.

INT 21h **Get/Set 386 | VMM Parameters**
AX = 252Eh **ver: 2.3**

This call is used only with 386 | VMM, to get or set virtual memory parameters. If made when 386 | VMM is not present, this call always returns success, and zeroes the entire parameter buffer if getting parameters.

INT 21h **Write Record to 386 | VMM Page Log File**
AX = 252Fh **ver: 3.0**

This call is used only with 386 | VMM, to create a type 3 record in the 386 | VMM page log file. If made when 386 | VMM is absent, this call has no effect.

INT 21h **Set DOS Data Buffer Size**
AX = 2530h **ver: 2.3**

Input:	ECX	= size of data buffer, in bytes. Must be between 1 KB and 64 KB.

Output:	If success:	
	Carry flag	= clear
	If failure:	
	Carry flag	= set
	EAX	= 8, if insufficient conventional memory
		= 129, if invalid size in ECX

This call is used to dynamically change the size of the data buffer used for DOS system calls. It is also used to reduce that buffer's size before performing an EXEC, to free up conventional memory for the child program. After the EXEC completes, the data buffer size should be increased again to allow file I/O to be performed efficiently.

The address of the MS-DOS data buffer always changes when its size is changed. Both its address and its size can be obtained with function 2517h, Get Info on DOS Data Buffer.

INT 21h AX = 2531h — Read/Write LDT Segment Descriptor

ver 3.0

Input:	BX	= segment selector	
	DS:EDX	= pointer to 8-byte buffer for descriptor contents	
	ECX	= 0, if reading LDT descriptor	
		= 1, if writing LDT descriptor	

Output:	If success:		
	Carry flag	= clear	
	If failure:		
	Carry flag	= set	
	EAX	= 129, if invalid segment selector	
		= 130, if under DPMI and not a code or data segment	

This call reads and writes LDT descriptors directly for programs (such as debuggers) that need to do their own segment management. The segment must be marked present, or invalid selector (error 129) is returned. Segments marked not present are always considered free (available to be allocated) by 386 | DOS-Extender.

An LDT descriptor is allocated by making an Allocate Segment call (INT 21h function 48h) with a size of zero.

When reading (but not when writing) a descriptor, any segment accessible to 386 | DOS-Extender (i. e., including GDT segments except under DPMI where the 386 | DOS-Extender itself runs out of the LDT and has no GDT access) is allowed.

Read access is allowed to 386 | DOS-Extender descriptors so that a debugger can examine privilege level zero code and data if an exception occurs inside a system call. For example, a debugger running at privilege level 3 can use the following steps to disassemble at the current CS:EIP after an exception, even if the CS is a level 0 segment:

- read the CS descriptor with this system call
- allocate an LDT descriptor with INT 21h function 48h
- modify the contents of the CS descriptor to be a data segment with DPL (Descriptor Privilege Level) 3, and use this system call to write it to the LDT descriptor
- use the LDT segment to disassemble the code (after checking to make sure EIP is not past the descriptor limit).

The same procedure can be used to examine segments pointed to by any segment registers at exception time.

When running under DPMI and writing the descriptor, bit 4 of the segment attributes byte must be set to 1, indicating that the segment is either a code or data segment (not a system segment or gate).

INT 21h AX = 2532h — Get Protected-Mode Processor Exception Vector

ver: 3.0

Input:	CL	= number of exception to get

Output:

If success:

Carry flag	= clear
ES:EBX	= exception handler address

If failure:

Carry flag	= set
EAX	= 129, if invalid exception number

This call is used to obtain the interrupt vectors for processor exceptions 00h-0Fh. Interrupt vectors 08h-0Fh carry dual meanings, covering hardware interrupts and processor exceptions. System calls 2502h, 2504h, and 2507h deal with hardware interrupts, and system calls 2532h and 2533h deal with processor exceptions. Please see Chapter 6 for a more complete discussion of interrupt processing.

For page fault handlers, please see also function 2522h.

INT 21h AX = 2533h — Set Protected-Mode Processor Exception Vector

ver: 3.0

Input:	CL	= number of exception to set
	DS:EDX	= address of exception handler

Output:

If success:

Carry flag	= clear

If failure:

Carry flag	= set
EAX	= 129, if invalid exception number

This call is used to set the interrupt vectors for processor exceptions 00h-0Fh. Interrupt vectors 08h-0Fh carry dual meanings, covering hardware interrupts and processor exceptions. System calls 2502h, 2504h, and 2507h deal with hardware interrupts, and system calls 2532h and 2533h deal with processor exceptions. Please see Chapter 6 for a more complete discussion of interrupt processing.

For page fault handlers, please see also function 2522h.

INT 21h — AX = 2534h — Get Interrupt Flag — ver: 3.0

Input:	None	
Output:	Carry flag	= clear
	EAX	= 0, if interrupts disabled
		= 1, if interrupts enabled

Applications that are DPMI-compatible cannot expect to get the correct interrupt flag with PUSHFD, and cannot modify the interrupt flag with POPFD or IRETD. Applications that manipulate the interrupt flag and run under DPMI must always use CLI/STI to modify the interrupt flag, and use this system call to obtain the current interrupt flag setting.

Please see section 6.7 for a more complete discussion of interrupts and the DPMI environment, and Appendix F for a discussion of DPMI compatibility.

INT 21h AX = 2535h — Read/Write System Registers

ver: 3.0

Input:	EBX	= 0, if reading system registers
		= 1, if writing system registers
	DS:EDX	= pointer to 48-byte buffer for system register values

Output: Carry flag = clear
If reading register values:
48 bytes of buffer at DS:EDX filled in

This call is used to read or write the debug registers, and the EM and MP bits of CR0. The format of the buffer at DS:EDX is:

0000h	CR0 value, only EM and MP bits are used (not available under DPMI 0.9)
0004h	DR0 value
0008h	DR1 value
000Ch	DR2 value
0010h	DR3 value
0014h	reserved
0018h	reserved
001Ch	DR6 value
0020h	DR7 value
0024h	reserved
0028h	reserved
002Ch	reserved

This call provides access to some of the processor's system registers. System registers cannot be read or written directly by programs running at privilege level 3. If you need access to other system registers, you must run at privilege level 0; see Appendix H. When reading the registers, all bits in CR0 except EM and MP are cleared. When writing the registers, the EM and MP bits are set as specified in CR0, and all other bits in CR0 are ignored.

For upward compatibility with future Intel processors, and with future versions of 386 | DOS-Extender, always perform the following steps when writing system registers:

1. Read the system registers into your buffer.
2. Modify only those bits you need to change in registers; preserve all bits that are undefined or that you don't need to change. Never modify any of the reserved areas in the buffer.
3. Write the system registers.

INT 21h AX = 2536h — Minimize/Maximize Extended/Conventional Memory Usage — ver: 3.0

```
Input:     EBX     = options
             bit 00000001h   = 0, if modifying extended memory usage
                             = 1, if modifying conventional memory
                                  usage
             bit 00000002h   = 0, if minimizing memory usage
                             = 1, if maximizing memory usage
           If minimizing memory usage:
             ECX  = number of pages above minimum to set new
                      limit to
           If maximizing memory usage:
             ECX  = number of pages below maximum to set new
                      limit to

Output:    If success:
             Carry flag    = clear
             EAX           = new limit, in pages, of
                               extended/conventional memory usage
           If failure:
             Carry flag    = set
             EAX           = 8, if memory error or -NOPAGE switch used
             EBX           = maximum limit, in pages
             ECX           = minimum limit, in pages
```

This call is used to restrict physical extended or conventional memory usage. Its functionality is the same as function 2521h (Limit Program's Extended Memory Usage) and function 2525h (Limit Program's Conventional Memory Usage), both of which take an absolute page limit.

This call can be more convenient because it takes a limit relative to the minimum or maximum possible limit. This means the caller need not ascertain what the acceptable range of limit values is before making the call. Please see the descriptions for 2521h and 2525h calls for more information.

INT 21h AX = 2537h — Allocate Conventional Memory Above DOS Data Buffer

ver: 3.0

Input:	BX	= number of paragraphs of conventional memory requested
Output:	If success:	
	Carry flag	= clear
	AX	= real-mode paragraph address of allocated memory block
	If failure:	
	Carry flag	= set
	AX	= 7, if MS-DOS memory control blocks corrupted
		= 8, if insufficient memory
	BX	= size, in paragraphs, of largest available memory block

This call allocates a conventional memory block above the buffer allocated for buffering data on DOS system calls. This is accomplished by moving the MS-DOS data buffer lower in the conventional memory space. Please see Figure 5-5 for a diagram showing how 386 | DOS-Extender manages conventional memory.

If a conventional memory block is allocated by calling MS-DOS (INT 21h function 25C0h), it will prevent either the size of the MS-DOS data buffer from being increased (if the block is allocated with the MS-DOS last fit allocation strategy), or the size of the application program memory buffer from being increased (if the block is allocated with the MS-DOS first fit allocation strategy). In many cases this side effect is unimportant (if the application is not planning to resize either buffer).

However, if the application needs to allocate conventional memory and also to retain the ability to resize both the MS-DOS data buffer and the application memory buffer, this call (2537h) should be used.

INT 21h **Get Registers from Interrupt State**
AX = 2538h **ver: 3.0**

Input:	DS:EBX	= pointer to 60-byte buffer	
	ECX	= 0, if get first interrupt state	
		= 1, if get next interrupt state	
	If ECX	= 1:	
	EDX		=handle for current interrupt state

Output:	If success:		
	Carry flag	= clear	
	register buffer at DS:EBX filled in		
	EDX	= handle for interrupt state	
	If failure:		
	Carry flag	= set	
	EAX	= 129, if invalid handle passed in EDX	
		= 131, if register saving not enabled	
		= 132, if no more interrupt states	

The registers are saved in the buffer in the following format:

Offset	Size	Description
0000h	8 BYTEs	unused, not modified
0008h	DWORD	EAX value
000Ch	DWORD	EBX value
0010H	DWORD	ECX value
0014h	DWORD	EDX value

Offset	Size	Description
0018h	DWORD	ESI value
001Ch	DWORD	EDI value
0020h	DWORD	EBP value
0024h	DWORD	ESP value
0028h	WORD	CS value
002Ah	WORD	DS value
002Ch	WORD	SS value
002Eh	WORD	ES value
0030h	WORD	FS value
0032h	WORD	GS value
0034h	DWORD	EIP value
0038h	DWORD	EFLAGS value

If interrupt state saving (function 2527h) is turned on, this call can be used to walk through interrupt states, from most recent to oldest, to get back to the state of most interest (e.g., to the interrupt that occurred when the application was running, if you are writing a code profiler).

Note if you are making this call from inside an interrupt handler, you will not see an interrupt state for the current interrupt, because your handler got control, rather than the 386 | DOS-Extender handler that saves registers. Your handler is responsible for getting the state from the current interrupt.

Please see also Get Last Protected Mode State after MS-DOS CTRL-C Interrupt (function 2528h).

INT 21h **Get .EXP Header Offset in Bound File**
AX = 2539h **ver: 3.0**

```
Input:      BX     = MS-DOS file handle for open file

Output:     If success:
              Carry flag   = clear
              EAX          = offset of .EXP header in file
              Current MS-DOS pointer in file modified
            If failure:
              Carry flag   = set
              Current MS-DOS pointer in file modified
              EAX          = 2 (file error)
              EBX          = 2, if MS-DOS error seeking in file
                           = 3, if MS-DOS error reading from file
                           = 4, if not a flat model .EXP or .REX file, or
                                MS-DOS .EXE file
                           = 5, if invalid file format
              If EBX       = 2 or 3 (MS-DOS error)
                ECX              = error code returned by MS-DOS,
                                     including:
                                 = 5, if access denied
                                 = 6, if invalid file handle
```

This call returns the offset of an application .EXP header, in bytes, in an .EXE file containing 386 | DOS-Extender (and optionally 386 | VMM) bound to the application program. It is intended for use by programs (such as debuggers) that need to read information from the header of an application .EXP file.

It is legal to make this call on an unbound file (including a real mode .EXE file); the returned offset is zero for unbound files.

INT 21h AX = 253Ah — Install Resize Segment Failure Handler — ver: 3.0

Input:	ES:EBX	= address of handler to call when INT 21h function 4Ah is about to return failure
Output:	Carry flag	= clear
	ES:EBX	= address of previous handler

This call installs a handler that is called when INT 21h function 4Ah (Resize Segment) is about to return failure. By default, there is no handler for this condition. You can remove an existing handler by passing in a null (zero) selector in ES.

The handler does not need to be locked down under 386 | VMM (i.e., it can cause page faults). The handler is called with a FAR CALL instruction, and must return with a FAR RETurn. The current stack is in the interrupt stack buffer that was allocated for the INT 21h interrupt handler. The current contents of all other registers are undefined. The handler must preserve all segment registers, but can modify any of the general registers. The handler returns a fail flag in EAX; if EAX contains one (true) the function 4Ah call is failed; if EAX contains zero (false) the call is retried (the segment resize is attempted again). If the handler requests a retry and the segment resize fails again, the handler will be called again (i.e., the handler is responsible for preventing infinite loops on retries).

When the handler gets control, the following information is passed on the stack:

	DWORD	offset (relative to current SS) of interrupt stack frame (please see section 6.5) for the INT 21h function 4Ah call
	DWORD	if error 8 (memory error), additional memory error code that will be set (see function 2519h)
	DWORD	error code that will be returned in EAX by the INT 21h function
	DWORD	requested new segment size, in pages
	DWORD	current segment size, in pages
	DWORD	selector for segment being resized
SS:ESP –>	QWORD	FAR return address from call

Resize Segment (function 4Ah) is used by most compiler run-time libraries to attempt to grow the memory heap when the application makes a memory allocate call and there is no memory available in the heap. For example, in a C program, the next thing that usually happens after INT 21h function 4Ah returns failure is the malloc() run-time library function returns NULL indicating no memory is available.

It is **not recommended** that applications install segment resize failure handlers. Correct behavior in an application program is to be able to handle a NULL return from a memory allocate call without crashing.

INT 21h — Jump to Real Mode
AX = 253Bh — **ver: 3.0**

Input: DS:EBX = pointer to 60-byte buffer containing real mode register values

Output: This call does not return

This call is used to jump to real mode without 386 | DOS-Extender allocating any data structures or buffers used to retain context for a return to protected mode. It is intended for use by programs, such as debuggers, that jump to a real mode state without returning. If interrupts are disabled when the call is made, they will remain disabled all the way through to the real mode entry point.

The format of the register buffer is shown below. The application is responsible for **all** real mode register values, including valid SS:ESP and EFLAGS.

Offset	Size	Description
0000h	8 BYTEs	unused, not modified by this call
0008h	DWORD	EAX value
000Ch	DWORD	EBX value
0010h	DWORD	ECX value
0014h	DWORD	EDX value
0018h	DWORD	ESI value
001Ch	DWORD	EDI value
0020h	DWORD	EBP value
0024h	DWORD	ESP value
0028h	WORD	CS value
002Ah	WORD	DS value
002Ch	WORD	SS value
002Eh	WORD	ES value
0030h	WORD	FS value
0032h	WORD	GS value
0034h	DWORD	EIP value
0038h	DWORD	EFLAGS value

To jump from real mode to protected mode, obtain the real mode address of a 386 | DOS-Extender routine from the Get Configuration Information system call (function 2526h). The jump to protected mode is performed by setting up the stack as shown below and executing a FAR CALL to the 386 | DOS-Extender jump to protected mode routine.

SS:SP –>	DWORD	real mode FAR pointer to register buffer (described above) containing protected mode register values

INT 21h **Shrink 386 | VMM Swap File**
AX = 253Ch **ver: 3.0**

This call is only used with 386 | VMM. It reduces the size of the swap file on disk, if possible. If this call is made when 386 | VMM is not present, it has no effect.

INT 21h **Allocate MS-DOS Memory Block**
AX = 25C0h **ver: 1.1w**

Input:	BX	= number of paragraphs of MS-DOS memory requested

Output:	If success:	
	Carry flag	= clear
	AX	= real-mode paragraph address of allocated memory block
	If failure:	
	Carry flag	= set
	AX	= error code
		= 7, if MS-DOS memory control blocks destroyed
		= 8, if insufficient memory
	BX	= size, in paragraphs, of largest available memory block

This function is used to allocate memory that can be accessed in real mode. Its most common usage is for mixed real- and protected-mode programs. By default, all conventional memory is used by 386 | DOS-Extender for the application program, and this call will not succeed. Conventional memory must explicitly be left free by the -MINREAL and/or -MAXREAL switches at program load time, or must be freed up at run time with the 2525h or 2536h system calls. See section 5.2.1 for more information.

INT 21h **Release MS-DOS Memory Block**
AX = 25C1h **ver: 1.1w**

Input:	CX	= real-mode paragraph address of memory block to free

Output:	If success:	
	Carry flag	= clear
	EAX	= destroyed
	If failure:	
	Carry flag	= set
	AX	= error code
		= 7, if MS-DOS memory blocks destroyed
		= 9, if invalid memory block address in CX

The conventional memory block previously allocated by function 25C0h is freed.

INT 21h AX = 25C2h	Resize MS-DOS Memory Block ver: 1.1w

Input:	BX	= new requested block size, in paragraphs
	CX	= real-mode paragraph address of memory block to modify

Output:	If success:	
	Carry flag	= clear
	EAX	= destroyed
	If failure:	
	Carry flag	= set
	AX	= error code
		= 7, if MS-DOS memory control blocks destroyed
		= 8, if insufficient memory
		= 9, if invalid memory block address in CX
	BX	= maximum size, in paragraphs, of memory block

This function attempts to increase or decrease the size of a conventional memory block previously allocated by a call to function 25C0h.

Note that some versions of MS-DOS have a bug that causes the block to be increased to its maximum possible size when error 8 (insufficient memory) is returned.

INT 21h **Execute Program**
AX = 25C3h **ver: 1.2b**

Input:	ES:EBX	= pointer to parameter block
	DS:EDX	= pointer to zero-terminated program file name

Output:

If success:
  Carry flag = clear
  All registers unchanged
If failure:
  Carry flag = set
  EAX = error code
    = 2, if file not found or path invalid
    = 5, if access denied
    = 8, if insufficient memory to load program
    = 10, if environment invalid
    = 11, if invalid file format
  All other registers unchanged

This function is used to load and execute other real-mode programs which will communicate with the parent protected-mode program. This call differs from system call 4Bh in that the A20 (address line 20) state is left disabled, and the 386 | VMM swap file is not flushed. Most applications should use the 4Bh form of EXEC. The 25C3h call exists primarily for historical reasons.

The format of the parameter block pointed to by ES:EBX is:

Offset	Size	Description
00h	DWORD	offset of environment string
04h	WORD	segment selector of environment string
06h	DWORD	offset of command tail string
0Ah	WORD	segment selector of command tail string

If a null (zero) segment selector is passed for the environment string, the parent's environment block is duplicated for the child. Otherwise, the environment string is a sequence of zero-terminated strings, with the whole block terminated by an extra zero byte.

The command tail string consists of a count byte, followed by an ASCII string terminated with a carriage return. The count value does not include the count byte itself or the terminating carriage return — it only accounts for the number of bytes in the ASCII string.

BIOS System Calls

BIOS system calls are made with a selection of software interrupts between 05h and 1Fh. Parameters are passed in registers. All registers, except those used to return results, are preserved across calls. This appendix lists all supported BIOS system calls.

Several of the BIOS system interrupts take a function number for the desired system function in register AH. Undocumented functions are passed through to the BIOS with all general registers unmodified and with the DS and ES segment registers destroyed, as described in section 6.1 of this manual.

Many undocumented system calls do not use segment registers; these can be issued directly from protected mode, using the standard real mode calling conventions. To use an undocumented system call that uses segment registers:

- use the -CALLBUFS switch to allocate a data buffer in conventional memory
- at run time, call INT 21h function 250Dh to obtain real mode and protected mode pointers to the buffer allocated with -CALLBUFS
- use INT 21h function 2511h to issue the desired interrupt in real mode, with the appropriate segment register set to the real mode address of the conventional memory buffer.

The BIOS system calls are not described in detail, and not all of the BIOS calls documented below are supported on all systems. For more information on a specific BIOS function, check the BIOS reference manual for your system.

TABLE C-1
BIOS SYSTEM CALL SUMMARY

Interrupt Number	Function Number	Function Name
05h	none	Print Screen
10h	00h	Set Video Mode
10h	01h	Set Cursor Type
10h	02h	Set Cursor Position
10h	03h	Read Cursor Position
10h	04h	Read Light Pen Position
10h	05h	Select Active Display Page
10h	06h	Scroll Up or Blank Display Window
10h	07h	Scroll Down or Blank Display Window
10h	08h	Read Character and Attribute at Cursor Position
10h	09h	Write Character and Attribute at Cursor Position
10h	0Ah	Write Character at Cursor Position
10h	0Bh	Set Color Palette
10h	0Ch	Write Pixel
10h	0Dh	Read Pixel
10h	0Eh	Write Character in TTY Mode
10h	0Fh	Read Video State
10h	10h	Color Control
10h	12h	Video Control
10h	13h	Write Character String
10h	1Ah	Get/Set Display Combination Code

(cont.)

Interrupt Number	Function Number	Function Name
10h	1Bh	Get Video State
10h	1Ch	Save/Restore Video State
10h	FEh	Get Video Buffer
10h	FFh	Update Video Buffer
11h	none	Equipment Configuration Check
12h	none	Get Conventional Memory Size
14h	00h	Initialize Communication Port
14h	01h	Output Character to Communications Port
14h	02h	Read Character from Communications Port
14h	03h	Get Communications Port Status
15h	84h	Read Joystick
15h	86h	Unconditional Wait
15h	88h	Get Size of Extended Memory
15h	C0h	Get System Environment
15h	C1h	Get Address of Extended BIOS Data Area
16h	00h	Read Character from Keyboard
16h	01h	Get Keyboard Status
16h	02h	Get Keyboard Codes
16h	03h	Set Repeat Rate
16h	05h	Place Character in Keyboard Buffer

(cont.)

Interrupt Number	Function Number	Function Name
16h	10h	Read Character from Enhanced Keyboard
16h	11h	Get Enhanced Keyboard Status
16h	12h	Get Enhanced Keyboard Flags
17h	00h	Write Character to Printer
17h	01h	Initialize Printer Port
17h	02h	Get Printer Status
19h	none	Reboot
1Ah	00h	Read Tick Counter
1Ah	01h	Set Tick Counter
1Ah	02h	Read Real Time Clock Time
1Ah	03h	Set Real Time Clock Time
1Ah	04h	Read Real Time Clock Date
1Ah	05h	Set Real Time Clock Date
1Ah	06h	Set Real Time Clock Alarm
1Ah	07h	Reset Real Time Clock Alarm
1Bh	none	CTRL-BREAK Handler
1Ch	none	Tick Counter
4Ah	none	Real Time Clock Alarm Service

INT 05h — Print Screen

Input:	None
Output:	None

This BIOS function prints the contents of the screen. Since this interrupt conflicts with the array bound processor exception, 386 | DOS-Extender relocates it to interrupt 80h (or to the interrupt specified with the -PRIVEC command line switch). Protected-mode code must issue an INT 80h to call this function. Real-mode code may issue either an INT 05h or an INT 80h to call this function. See section 6.8.3 and the description of system call 250Ch for more details.

INT 10h — Video Driver Services

Calls to INT 10h are used to request services from the BIOS video driver routines that program the video display adapter in the system.

INT 10h AH = 00h — Set Video Mode

Input:	AL	= mode
Output:	None	

Sets the current video display mode on the video controller. 386 | DOS-Extender will automatically update LDT segment 001Ch, which maps the video memory, if a call to this function is made from protected mode.

INT 10h
AH = 01h **Set Cursor Type**

Input:	CH <0-4>	= start scan line for cursor
	CL <0-4>	= end scan line for cursor
Output:	None	

Selects the scan lines to be used for the cursor display.

INT 10h
AH = 02h **Set Cursor Position**

Input:	BH	= page number
	DH	= row coordinate
	DL	= column coordinate
Output:	None	

Moves the cursor to the specified zero-relative character position on the screen (i.e., row 0, column 0 is the upper left character position).

INT 10h AH = 03h — Read Cursor Position

Input:	BH	= page number
Output:	CH <0-4>	= start scan line for cursor
	CL <0-4>	= end scan line for cursor
	DH	= character row coordinate
	DL	= character column coordinate

Returns the current cursor type and location on the display.

INT 10h AH = 04h — Read Light Pen Position

Input: None

Output:

If light pen triggered:

AH	= 1
BX	= pixel column coordinate
CH	= pixel row coordinate
DH	= character row coordinate
DL	= character column coordinate

If light pen not triggered:

AH	= 0
BX	= destroyed
CX	= destroyed
DX	= destroyed

Returns the status and position of the light pen.

INT 10h
AH = 05h

Select Active Display Page

Input:	AL	= page number of display page to make active
Output:	None	

Selects the display page to make active.

INT 10h
AH = 06h

Scroll Up or Blank Display Window

Input:	AL	= number of lines to scroll
		= 0, if the window is to be blanked
	BH	= attribute to use for blank lines
	CH	= upper-left row coordinate of window
	CL	= upper-left column coordinate of window
	DH	= lower-right row coordinate of window
	DL	= lower-right column coordinate of window
Output:	None	

Scrolls up or blanks the specified window on the display.

INT 10h
AH = 07h
Scroll Down or Blank Display Window

Input:	AL	= number of lines to scroll
		= 0, if the window is to be blanked
	BH	= attribute to use for blank lines
	CH	= upper left row coordinate of window
	CL	= upper left column coordinate of window
	DH	= lower right row coordinate of window
	DL	= lower right column coordinate of window
Output:	None	

Scrolls down or blanks the specified window on the display.

INT 10h
AH = 08h
Read Character and Attribute at Cursor Position

Input:	BH	= display page
Output:	AH	= attribute byte
	AL	= ASCII character value

Returns the ASCII character and its attribute byte at the current cursor position.

INT 10h AH = 09h — Write Character and Attribute at Cursor Position

Input:	AL	= ASCII character value
	BH	= display page
	BL	= attribute byte
	CX	= number of characters to write
Output:	None	

Writes the specified character and attribute to the current cursor position on the screen, replicating the character by the count specified in the CX register.

INT 10h AH = 0Ah — Write Character at Cursor Position

Input:	AL	= ASCII character value
	BH	= display page
	BL	= color attribute, if graphics mode
	CX	= number of characters to write
Output:	None	

Writes the specified character to the current cursor position on the screen, replicating the character by the count specified in the CX register and leaving the current character attribute unchanged.

INT 10h AH = 0Bh — Set Color Palette

Input:	BH	= color palette ID
	BL	= color value to be used with color ID
Output:	None	

Selects the color to be generated by the color palette for a specific color ID.

INT 10h AH = 0Ch — Write Pixel

Input:	AL	= pixel color value
	BH	= page
	CX	= pixel column coordinate
	DX	= pixel row coordinate
Output:	None	

Sets the color value of the specified pixel position on the screen, when in graphics mode.

INT 10h AH = 0Dh — Read Pixel

Input:	BH	= page
	CX	= pixel column coordinate
	DX	= pixel row coordinate
Output:	AL	= pixel color value

Returns the color value of the specified pixel position on the screen, when in graphics mode. The page number in BH is ignored for display modes that only support one page.

INT 10h AH = 0Eh — Write Character in TTY Mode

Input:	AL	= ASCII character value
	BH	= page
	BL	= foreground color, if graphics mode
Output:	None	

Writes the specified character to the current cursor position on the screen, processing ASCII teletype control characters appropriately.

INT 10h
AH = 0Fh **Read Video State**

Input:	None	
Output:	AH	= number of character columns on the screen
	AL	= current display mode
	BH	= active display page

Returns the current screen width, display mode, and active display page for the video controller.

INT 10h AH = 10h — Color Control

Input:

If setting palette register:

AL	= 00h
BH	= color value
BL	= palette register to set

If setting border color register:

AL	= 01h
BH	= color value

If setting all palette registers and the border color register:

AL	= 02h
ES:EDX	= pointer to 17-byte list of color values

If toggling blink-intensity bit:

AL	= 03h
BL	= 0, to toggle intensity bit
	= 1, to toggle blink bit

If getting palette register:

AL	= 07h
BL	= palette register

If getting border color:

AL	= 08h

If getting palette and border:

AL	= 09h
ES:EDX	= pointer to 17-byte buffer

If setting color register:

AL	= 10h
BX	= color register
CH	= green value
CL	= blue value
DH	= red value

If setting block of color registers:

AL	= 12h
BX	= first color register
CX	= number of color registers
ES:EDX	= pointer to buffer, 3 bytes/color register

If setting color page state:

AL	= 13h

If setting paging mode:

BL	= 00h

```
      BH        = paging mode
    If selecting color register page:
      BL        = 01h
      BH        = page
  If getting color register:
    AL        = 15h
    BX        = color register
  If getting block of color registers:
    AL        = 17h
    BX        = first color register
    CX        = number of color registers
    ES:EDX    = pointer to buffer, 3 bytes/color register
  If getting color page state:
    AL        = 1Ah
  If setting gray-scale values:
    AL        = 1Bh
    BX        = first color register
    CX        = number of color registers
```

Output:

```
  Function 07h:
    BH   = color
  Function 08h:
    BH   = color
  Function 09h:
    Buffer at ES:EDX filled in with color values
  Function 15h:
    CH   = green value
    CL   = blue value
    DH   = red value
  Function 17h:
    Buffer at ES:EDX filled in with register values
  Function 1Ah:
    BH   = color page
    BL   = paging mode
```

This function controls color registers.

INT 10h
AH = 12h

Video Control

Input:

If getting configuration information:

BL	= 10h

If selecting alternate printscreen:

BL	= 20h

If setting scan lines:

BL	= 30h
AL	= scan line code

If enabling/disabling default palette loading:

BL	= 31h
AL	= 00h, to enable
	= 01h, to disable

If enabling/disabling video:

BL	= 32h
AL	= 00h, to enable
	= 01h, to disable

If enabling/disabling gray-scale summing:

BL	= 33h
AL	= 00h, to enable
	= 01h, to disable

If enabling/disabling cursor emulation:

BL	= 34h
AL	= 00h, to enable
	= 01h, to disable

If switching active display:

BL	= 35h
AL	= switching functions

If AL = 0, 2, or 3:

ES:EDX	= pointer to 128-byte buffer

If enabling/disabling screen refresh:

BL	= 36h
AL	= 00h, to enable
	= 01h, to disable

Output:

Function 10h:

BH	= display type
BL	= memory install code

```
  CH   = feature bits
  CL   = switch setting
Function 30h:
  AL   = 12h, if VGA active
       = 00h, if VGA not active
Functions 31h, 32h, 33h, 34h, 36h:
  If function supported:
    AL      = 12h
Function 35h:
  If function supported:
    AL      = 12h
    If called with AL  = 0 or 2:
        Adapter state information saved in buffer at ES:EDX
    If called with AL  = 3:
        Adapter state loaded from buffer at ES:EDX
```

This function controls video parameters.

INT 10h **Write Character String**
AH = 13h

Input:	AL	= string format code
	BH	= display page
	BL	= attribute
	ECX	= number of bytes in string
	DH	= row coordinate to write string to
	DL	= column coordinate to write string to
	ES:EBP	= pointer to string to write

Output: None

Writes the specified string to the specified location on the screen. The size of the data cannot exceed the size of the 386 | DOS-Extender data buffer allocated under the control of the -MINIBUF and -MAXIBUF command line switches. If the data are too large to be buffered on a single call to the BIOS, this function will truncate the data to the size of the data buffer.

INT 10h AH = 1Ah — Get/Set Display Combination Code

Input:	AL	= 00h, if get display code
		= 01h, if set display code
	If AL	= 01h:
	BH	= inactive display code
	BL	= active display code
Output:	If function supported:	
	AL	= 1Ah
	If getting display code:	
	BH	= inactive display code
	BL	= active display code

This function gets or sets the display combination code.

INT 10h AH = 1Bh — Get Video State

Input:	BX	= implementation type
	ES:EDI	= pointer to 64-byte buffer
Output:	If function supported:	
	AL	= 1Bh
	Buffer at ES:EDI filled with state information	

This function returns the current video state.

INT 10h AH = 1Ch — Save/Restore Video State

Input:	AL	= 00h, to get state buffer size
		= 01h, to save state
		= 02h, to restore state
	CX	= requested states
	ES:EBX	= pointer to buffer

Output:

If function supported:
  AL = 1Ch
  If getting state buffer size:
    BX = buffer block count (64 bytes/block)
  If saving state:
    Buffer at ES:EBX filled with video state
  If restoring state:
    Video state loaded with information at ES:EBX

This function saves or restores the video state.

INT 10h AH = FEh — Get Video Buffer

Input:	EDI	= offset in first megabyte of physical video buffer
Output:	ES:EDI	= pointer to shadow video buffer

This function is used when executing under DESQview 386 to obtain the address of the shadow video buffer set up for the window. The segment selector returned in ES is always 0034h, the segment that maps the MS-DOS memory partition.

INT 10h AH = FFh — Update Video Buffer

Input:	ES:EDI	= pointer to first modified character in shadow video buffer
	CX	= number of characters to flush
Output:	None	

This function is used when executing under DESQview 386 to force some or all of the contents of the shadow video buffer to be flushed to the physical video buffer.

INT 11h — Equipment Configuration Check

Input:	None	
Output:	EAX	= equipment configuration

This function returns the hardware equipment configuration of the system.

INT 12h — Get Conventional Memory Size

Input:	None	
Output:	AX	= number of kilobytes of conventional memory

This function returns the number of KBs of conventional memory (memory below 640 KBs) installed on the machine.

INT 13h — Floppy Disk and Fixed Disk I/O

Calls to INT 13h are used to perform both floppy disk drive and fixed disk drive I/O. Drive numbers between 00h and 7Fh indicate floppy disk I/O; drive numbers between 80h and FFh indicate fixed disk I/O. On most systems, the only drive numbers supported by the BIOS are drive 00h for floppy disk drive A, drive 01h for floppy disk drive B, drive 80h for fixed disk drive C, and drive 81h for fixed disk drive D.

When both a fixed disk and a floppy disk drive are installed on the system, the BIOS redirects floppy disk drive operations (determined by examining the drive number) to INT 40h. Protected-mode programs running under 386 | DOS-Extender should **never** call INT 40h directly; they should always use INT 13h for floppy disk I/O.

Because the BIOS disk I/O functions were never really standardized by hardware manufacturers, 386 | DOS-Extender does not support them . Any functions which do not require pointers to data buffers work as specified in the BIOS reference manual. Any functions that require pointers cannot be called directly from protected mode. Instead, system call 2511h must be used to issue a real-mode interrupt with segment registers specified.

INT 14h — Serial Communications I/O

Calls to INT 14h are used to request BIOS system services for serial communications I/O.

INT 14h AH = 00h — Initialize Communication Port

Input:	AL	= port parameters
	DX	= port number
Output:	AH	= port status
	AL	= modem status

Initializes the communications port to the specified configuration.

INT 14h AH = 01h — Output Character to Communications Port

Input:	AL	= character to output
	DX	= port number
Output:	AH	= port status

Outputs the specified character to the specified communications port.

INT 14h AH = 02h — Read Character from Communications Port

Input:	DX	= port number
Output:	AH	= port status
	AL	= character read

This function returns the character read from the specified communications port.

INT 14h **Get Communications Port Status**
AH = 03h

Input:	DX	= port number
Output:	AH	= port status
	AL	= modem status

This function returns the status for the specified communications port.

INT 15h **System Services**

Calls to INT 15h are used to request functions that were added to the BIOS when the IBM PC AT systems were introduced, or that are specific to a particular system.

INT 15h **Event Wait**
AH = 83h

This function is not supported, as it requires a pointer to a semaphore in user memory, and user memory in a protected-mode program may not be conventional memory which can be accessed by the BIOS. Programs that need to use this function must have a real-mode procedure that makes the system call.

INT 15h AH = 84h — Read Joystick Switches

Input:	DX	= 0000h
Output:	AH	= 00h
	AL	= switch settings

Returns the settings for the joystick switches.

INT 15h AH = 84h — Read Joystick Potentiometer

Input:	DX	= 0001h
Output:	AX	= A(x) potentiometer value
	BX	= A(y) potentiometer value
	CX	= B(x) potentiometer value
	DX	= B(y) potentiometer value

Returns the joystick potentiometer values.

INT 15h AH = 86h — Unconditional Wait

Input:	CX	= high 16 bits of count of number of microseconds
	DX	= low 16 bits of count of number of microseconds

Output: If wait was performed:
  Carry flag = clear
If wait was not performed (because an Event Wait was active):
  Carry flag = set

Waits the specified number of microseconds, then returns.

INT 15h AH = 87h — Block Move

This function is not supported. Its purpose is to allow real-mode programs to access extended memory; this can be done directly by protected-mode programs.

INT 15h AH = 88h — Get Size of Extended Memory

Input:	None	
Output:	AX	= size in kilobytes of extended memory

This function returns the number of KBs of memory available in extended memory (memory above one MB). While the original BIOS handler for this call returns the amount of physical extended memory, this call is used by many programs, including 386 | DOS-Extender, for memory allocation, so the returned value may be less than the amount of physical memory actually on the system . See section 5.2.2 for more information.

INT 15h AH = 89h — Switch to Protected Mode

This function is not supported. Programs executing under 386 | DOS-Extender are already in protected mode.

INT 15h AH = C0h — Get System Environment

Input:	None	
Output:	ES:EBX	= pointer to a data buffer containing the system configuration parameters

This call returns a pointer to a data buffer containing the system configuration parameters.

INT 15h AH = C1h — Get Address of Extended BIOS Data Area

Input:	None	
Output:	ES:EAX	= pointer to data area

This function returns a pointer to the extended BIOS data area.

INT 16h — Keyboard I/O

This call is used to request keyboard I/O services from the BIOS routines that program the keyboard controller.

INT 16h AH = 00h — Read Character from Keyboard

Input:	None	
Output:	AH	= scan code
	AL	= character read

This function reads a character from the keyboard, waiting for one to be available if necessary.

INT 16h AH = 01h — Get Keyboard Status

Input: None

Output:
If keystroke waiting to be read:

Zero flag	= clear
AH	= scan code
AL	= character available

If keyboard input buffer empty:

Zero flag	= set

This function returns keyboard status and the next character waiting to be read, if one is available.

INT 16h AH = 02h — Get Keyboard Codes

Input:	None	
Output:	AL	= keyboard codes

This function returns the status of the shift keys on the keyboard (Shift, CTRL, ALT, etc.).

INT 16h AH = 03h — Set Repeat Rate

Input:	AL	= 05h
	BH	= repeat delay
	BL	= repeat rate
Output:	None	

This function sets the key repeat rate.

INT 16h
AH = 05h

Place Character in Keyboard Buffer

Input:	CH	= scan code
	CL	= character

Output:
If success:
- Carry flag = clear
- AL = 00h

If failure:
- Carry flag = set
- AL = 01h

This function places a character and a scan code in the keyboard type-ahead buffer.

INT 16h
AH = 10h

Read Character from Enhanced Keyboard

Input: None

Output:	AH	= scan code
	AL	= ASCII character

This function reads a character and a scan code from the keyboard type-ahead buffer.

INT 16h AH = 11h — Get Enhanced Keyboard Status

Input:	None
Output:	If keystroke in buffer: zero flag = clear AH = scan code AL = character If no key available: zero flag = set

This function checks for keystroke available, returns it if so.

INT 16h AH = 12h — Get Enhanced Keyboard Flags

Input:	None
Output:	AX = flags

This function returns keyboard status flags.

INT 17h — Printer I/O

This call is used to request printer I/O services from the BIOS routines that program the parallel printer I/O ports.

INT 17h AH = 00h — Write Character to Printer

Input:	AL	= character to write
	DX	= printer number
Output:	AH	= printer status

This function writes the specified character to the specified printer.

INT 17h AH = 01h — Initialize Printer Port

Input:	DX	= printer number
Output:	AH	= printer status

This function initializes the specified printer port.

INT 17h AH = 02h — Get Printer Status

Input:	DX	= printer number
Output:	AH	= printer status

This function returns the status for the specified printer port.

INT 19h — Reboot

Input:	None
Output:	None

This function reboots the machine. It is also taken over by programs that follow the VDISK standard for allocating extended memory. See section 5.2.2 for more information on this standard.

INT 1Ah — Real Time Clock Services

This call is used to request real time clock or system timer services from the BIOS.

INT 1Ah AH = 00h — Read Tick Counter

Input:	None	
Output:	AL	= 1, if 24 hours have elapsed since last call to this function
	CX	= high 16 bits of count
	DX	= low 16 bits of count

This function returns the contents of the tick counter.

INT 1Ah AH = 01h — Set Tick Counter

Input:	CX	= high 16 bits of count
	DX	= low 16 bits of count
Output:	None	

This function loads the tick counter with the specified value.

INT 1Ah AH = 02h — Read Real Time Clock Time

Input:	None	
Output:	CH	= BCD hours
	CL	= BCD minutes
	DH	= BCD seconds
	DL	= 01h, if daylight savings time
		= 00h, if not daylight savings time

This function returns the real time clock time.

INT 1Ah AH = 03h — Set Real Time Clock Time

Input:	CH	= BCD hours
	CL	= BCD minutes
	DH	= BCD seconds
	DL	= 01h, if daylight savings time
		= 00h, if not daylight savings time
Output:	None	

This function sets the real time clock time to the specified value.

INT 1Ah AH = 04h — Read Real Time Clock Date

Input:	None

Output:	CH = BCD century
	CL = BCD year
	DH = BCD month
	DL = BCD day

This function returns the real time clock date.

INT 1Ah AH = 05h — Set Real Time Clock Date

Input:	CH = BCD century
	CL = BCD year
	DH = BCD month
	DL = BCD day

Output:	None

This function sets the real time clock date to the specified value.

INT 1Ah AH = 06h — Set Real Time Clock Alarm

Input:	CH	= BCD hours
	CL	= BCD minutes
	DH	= BCD seconds

Output:	If alarm already set:	
	Carry flag	= set
	If success:	
	Carry flag	= clear

This function loads the specified values into the real time clock alarm register.

INT 1Ah AH = 07h — Reset Real Time Clock Alarm

Input:	None
Output:	None

This function resets the real time clock alarm register.

INT 1Bh — CTRL-BREAK

This interrupt is issued in real mode by the BIOS whenever CTRL-BREAK is pressed by the user. Programs that wish to be notified when this occurs may take over this interrupt either to gain control in a real-mode handler (with 386 | DOS-Extender system function 2505h), or to gain control in a protected-mode handler (with 386 | DOS-Extender system function 2506h).

INT 1Ch — Tick Counter

This interrupt is issued in real mode by the BIOS whenever the timer tick hardware interrupt occurs. Programs that wish to be notified when this occurs may take over this interrupt either to gain control in a real-mode handler (with 386 | DOS-Extender system function 2505h), or to gain control in a protected-mode handler (with 386 | DOS-Extender system function 2506h).

INT 1Dh — Pointer to Video Parameter Table

The real-mode vector for this interrupt (but **not** the protected-mode vector) points to a table in conventional memory containing the parameters for the video display adapter.

INT 1Eh — Pointer to Floppy Disk Parameter Table

The real-mode vector for this interrupt (but **not** the protected-mode vector) points to a table in conventional memory containing the parameters used by the system floppy disk controller.

INT 1Fh — Pointer to Graphics Characters Dot Table

The real-mode vector for this interrupt (but **not** the protected-mode vector) points to a user-supplied table in conventional memory containing dot graphics characters used by the BIOS in graphics video modes.

INT 40h — Floppy Disk I/O

For systems which have both a fixed and a floppy disk, the BIOS reissues INT 13h system calls as INT 40h for floppy disk I/O. Protected-mode programs running under 386 | DOS-Extender should always call INT 13h for floppy disk system services, never INT 40h.

INT 41h — Pointer to Fixed Disk Drive 1 Parameter Table

The real-mode vector for this interrupt (but **not** the protected-mode vector) points to a table in conventional memory containing some of the parameters used by the system fixed disk drive controller.

INT 46h — Pointer to Fixed Disk Drive 2 Parameter Table

The real-mode vector for this interrupt (but **not** the protected-mode vector) points to a table in conventional memory containing some of the parameters used by the system fixed disk drive controller.

INT 4Ah — Real Time Clock Alarm Service

This interrupt is issued in real mode by the BIOS whenever the alarm function of the real time clock is enabled and the alarm time has been reached. Programs that wish to be notified when this occurs may take over this interrupt either to gain control in a real-mode handler (with 386 | DOS-Extender system function 2505h), or to gain control in a protected-mode handler (with 386 | DOS-Extender system function 2506h).

Libraries, Header Files, and Example 80386 Programs

The product distribution disks include:

☛ C-callable libraries of routines for 386|DOS-Extender system calls, and some MS-DOS system calls

☛ C and assembly language header files defining constants and data structures for 386|DOS-Extender

☛ example 80386 code demonstrating basic concepts and a number of common programming techniques

This appendix briefly describes the example code on the distribution disks. Since file organization on product disks can be changed after the documentation has gone to print, please refer to the product installation instructions if you have trouble finding particular files.

You may incorporate object code forms of the example programs and the DOS and 386 | DOS-Extender system call libraries in your own programs. You may not redistribute any of the source code to the example programs with your own programs.

D.1 The \EXAMPLES\INCLUDES Directory

This directory contains C and assembly language header files defining constants and data structures for DOS and 386 | DOS-Extender system calls, and function prototypes for the libraries in the \EXAMPLES\LIBS directory. It includes:

PLTYPES.H	Simple data types and basic constants, macros to construct and break apart protected-mode and real-mode FAR pointers.

PHARLAP.H	Constants and data structures for 386 | DOS-Extender and DOS system calls, and function prototypes for the DOSX32.LIB and DOS32.LIB libraries.
DOSX.AH	Assembly language constants and data structures for 386 | DOS-Extender and DOS system calls.
PLEXP.H	Structure of an .EXP file header.
HW386.H, HW386.AH	C and assembly language constants and data structures for 80386/80486 register definitions and system data structures, such as segment descriptors and task state segments.

D.2 The \EXAMPLES\LIBS Directory

This directory contains two libraries of C-callable functions:

DOS32.LIB	Selected DOS system calls.
DOSX32.LIB	Selected 386 | DOS-Extender system calls.

The function prototypes are defined in the PHARLAP.H file in the INCLUDES directory. The following calling conventions are used:

- all parameters are passed on the stack
- parameters are always pushed as a multiple of a DWORD (4 bytes), to keep the stack DWORD-aligned
- parameters are pushed in reverse order (first parameter is the last one pushed)
- the caller is responsible for popping parameters off the stack
- all registers except EAX are preserved
- all functions return an error code in EAX, zero for no error.

D.3 The \EXAMPLES\DOSEXT Directory

The example programs in the \EXAMPLES\DOSEXT directory illustrate the basic structure of a protected-mode application program that makes system calls.

Each program is distributed as two files:

- an assembly language source code file, with a file name extension of .ASM
- an MS-DOS batch file to assemble and link the program, with a file name extension of .BAT.

RMHELLO	A short real-mode program that uses 80386 instructions and 32-bit instruction operands.
HELLO	A short application which prints out the string "Hello world" on the screen. This example illustrates how to make MS-DOS system calls.
PECHO	A program which echoes its command line arguments to the screen. This example illustrates how to access the MS-DOS command line from a protected-mode program. For example, the command: RUN386 PECHO TESTING 1 2 3 displays "TESTING 1 2 3" on the screen.
SCRTEST	This program demonstrates how to access directly the physical screen memory from a protected-mode program, using segment selector 001Ch. The program clears the screen and displays the PC character set on the screen in a banner box.
DOSDEMO	This program performs a variety of MS-DOS system calls from protected mode. It makes screen, keyboard, and file I/O system calls.

TAIL	The mixed real- and protected-mode program that uses the Microsoft mouse and is discussed in section 7.9.2.
PTAIL2, RTAIL2	A mixed real- and protected-mode program that is implemented as two separate programs and which has the same functionality as the TAIL program. This program contains examples of switching modes with software interrupts, and of using the intermode call buffer that is allocated with the -CALLBUFS command line switch.

D.4 The \EXAMPLES\GRAPHICS Directory

The example programs in the GRAPHICS directory illustrate calling real-mode libraries from a protected-mode program. This example is also described in section 7.9.1. To build this program, you need both the MetaWare High C-386 compiler and the Microsoft C 5.1 compiler.

README	Describes the program architecture and how to build the program.
GSERVER.H	A C header file that defines the communication protocol between the real- and protected-mode code.
GDEMO.C	The main protected-mode C program that makes calls to real-mode Microsoft C graphics routines.
GSERVER.C	The real-mode Microsoft C program that links in the graphics library.
GGLUE.C, PROT.ASM	Protected-mode glue code for handling cross-mode function calls.
REAL.ASM	Real-mode glue code for handling cross-mode function calls.
GDEMO.MAK	A make file to build the protected-mode main program.

GSERVER.MAK	A make file to build the real-mode program with the Microsoft C graphics routines.

D.5 The \EXAMPLES\INTHNDLR Directory

This directory contains the example interrupt handlers described in Appendix E. It includes the following sample programs:

ICHAIN.ASM	Demonstrates how to chain to a previous interrupt handler for INT 21h.
ERR387.ASM	Demonstrates processing a hardware interrupt (IRQ13, the coprocessor interrupt) and returning.
CTRLC.ASM	Demonstrates how to correctly call C code from an interrupt handler for the MS-DOS CTRL-C interrupt.
SERIO.ASM	Demonstrates how to write bimodal handlers (one real-mode and one protected-mode handler) to service a hardware interrupt (IRQ4, serial port 1) that occurs frequently.
CRITERR.ASM	Demonstrates how to find a real-mode stack frame from a protected-mode handler for the MS-DOS critical error interrupt.

Example Interrupt Handlers

This appendix describes the sample interrupt handlers in the \EXAMPLES\INTHNDLR directory on the distribution disks noted in Appendix D. For each sample program, the portion of the program demonstrating a particular technique is reproduced and elaborated with a supporting text explanation. These sample programs can all run at both privilege level zero (-PRIVILEGED) and level three (-UNPRIVILEGED).

1. ICHAIN.ASM: An assembly language program that includes I21_FCN9_HND, a replacement for the INT 21h (Function 09h) handler. It shows how to install a replacement routine for a particular interrupt and a method for chaining to the previous handler for functions not included in your replacement.

2. ERR387.ASM: An assembly language program that includes IRQ13_HNDLR, a protected-mode coprocessor exception handler. This file demonstrates how to write and install a simple handler for hardware-generated interrupts.

3. CTRLC.ASM: An assembly language program with a handler for the MS-DOS CTRL-C interrupt. It includes CC_HNDLR, a protected-mode CTRL-C handler that demonstrates setting up a stack in the main program data segment so that it is possible to call a C routine.

4. SERIO.ASM: A demo program for handling IRQ4 (COM1) hardware interrupts. This program demonstrates the technique of installing bimodal interrupt handlers (that is, separate real- and protected-mode interrupt handlers that do the same thing). This is desirable when processing interrupts that occur very frequently. Because the program can handle the interrupt in either mode, no extra overhead for switching between real and protected mode is incurred.

5. CRITERR.ASM: An assembly language program that includes CRITE_HND, a protected-mode handler for the MS-DOS critical error interrupt. It demonstrates how to access a real-mode interrupt stack frame from a protected-mode handler.

ICHAIN.ASM

The ICHAIN.ASM example program installs an INT 21h handler that chains to the original INT 21h handler for most functions. For function 09h (print string), it uses segment 001Ch (the screen buffer segment) to display the string, rather than letting MS-DOS display the string. This program demonstrates how to chain to a previous interrupt handler either after performing your own processing, or in cases in which no processing is required.

The ICHAIN program prints the following output:

```
Demo program (take over INT 21h, for func 09h processing)
     This string printed with standard INT 21h handler

This string printed by our INT 21h handler

This string printed with INT 21h func 02h
```

where the line in bold is printed high intensity on the display. The actual INT 21h interrupt handler routine from ICHAIN.ASM is reproduced below:

```
;*****************************************************************************
; I21_FCN9_HND - Replacement INT 21h Handler.  Chains to
;       original handler for any function except 09h (print string).
;
;       For function 09h, prints the string using segment 001Ch (the screen
;       segment), and also uses the high intensity attribute to show it was
;       printed by this handler and not by MS-DOS.
;
; Note: This handler ASSUMES an 80-column text display video mode, page 0
;*****************************************************************************
        public  i21_fcn9_hnd
i21_fcn9_hnd    proc    far
        pushfd                          ; save flags in case of chain
        cmp     ah, 09h                 ; branch if function 09h
        je      short #func_09                  ;
```

```
;
; Chain to original INT 21h handler, with all registers and flags preserved
;
        sub     esp, 8                  ; save space for IRETD frame
        push    ds                      ; save registers we use
        push    eax                     ;
        mov     ax, SS_DATA             ; set DS to our data segment
        mov     ds, ax                  ;
        mov     eax, prhnd_off          ; put original handler addr in IRETD
        mov     [esp+8], eax            ; frame
        movzx   eax, prhnd_sel          ;
        mov     [esp+12], eax           ;
        pop     eax                     ; restore registers
        pop     ds                      ;
        iretd                           ; chain to prev handler

#func_09:
;
; Print the string at DS:EDX to the display, using segment 001Ch, and
; turning on the high intensity video attribute
;
        assume  ds:nothing
        sti                             ; re-enable interrupts
        add     esp,4                   ; pop unneeded flags off stack
        push    eax                     ; save registers we use
        push    ebx                     ;
        push    ecx                     ;
        push    edx                     ;
        push    es                      ;
        push    edx                     ; save string pointer
        mov     ah, 03h                 ; get cursor position in page 0
        mov     bh,0                    ;
        int     10h                     ;
        xor     ebx,ebx                 ; convert to offset in screen seg
        mov     bl,dh                   ; (assume 80 cols/row, page 0)
        imul    ebx,160                 ;
        and     edx,0FFh                ;
        add     ebx,edx                 ;
        pop     edx                     ; restore string pointer
        mov     ax,SS_SCREEN            ; set ES:EBX to screen position
        mov     es,ax                   ;
        mov     ah,0Fh                  ; set high intensity video attribute
#loop:
        mov     al,[edx]                ; get next char
        cmp     al,'$'                  ; branch if no more in string
        je      short #exit             ;
        mov     es:[ebx],ax             ; store char in video buffer
        inc     edx                     ; bump ptrs & continue loop
        add     ebx,2                   ;
        jmp     #loop                   ;
```

```
#exit:
        pop     es                          ; restore regs
        pop     edx                                 ;
        pop     ecx                                 ;
        pop     ebx                                 ;
        pop     eax                                 ;
        iretd                               ; return to interrupted code
i21_fcn9_hnd    endp
```

At the beginning of the routine, if the function is not 09h, the code chains to the original interrupt handler (whose address was previously stored in a global variable), preserving all registers and flags. This is done by:

- reserving 12 bytes for an interrupt return (IRETD) stack frame
- saving any needed registers on the stack
- placing the original EFLAGS and the CS:EIP of the original handler in the IRETD frame
- restoring original register values and executing an IRETD to chain to the previous handler.

For function 09h, the routine just saves all needed registers, uses the screen segment (001Ch) to print the string in high intensity, and finally restores registers and returns from the interrupt. Because this is a simple example, the print screen code assumes an 80-column text video mode, page 0, and does not process the carriage return and line feed characters.

ERR387.ASM

The ERR387.ASM example program installs a protected mode IRQ13 (the 387 coprocessor exception interrupt) handler and then causes a 387 exception by taking the square root of a negative number. The interrupt handler clears the source of the interrupt and prints out the coprocessor environment. This program demonstrates how to write a simple protected mode hardware interrupt handler.

The ERR387 program prints the following output:

```
80387 environment block:
00000040
0000B881
00003FFF
0000005E
0000000C
00000180
00000000
```

The IRQ13 hardware interrupt handler from the program follows:

```
;****************************************************************************
; IRQ13_HNDLR - Protected Mode Coprocessor Exception Handler
;       This handler just stores the coprocessor environment and prints it
;       out, clears the source of the exception, and returns.
;****************************************************************************
        public  irq13_hndlr
irq13_hndlr proc        near
;
; Save regs and re-enable interrupts.  It's good to re-enable interrupts
; when possible in an interrupt handler, so other hardware interrupts can
; occur.
;
        sti                             ; re-enable interrupts
        push    eax                     ; save all registers we modify
        push    ds                              ;

;
; Acknowledge the interrupt.  Until we do this, neither this interrupt
; (IRQ13) nor any lower priority hardware interrupt can occur.  After
; we do this, we must be prepared to deal with being reentered.  In the
; case of a coprocessor exception handler, we don't have to worry about
; reentrancy because we won't cause another coprocessor exception inside
; the handler.
;
        mov     al, 20h                 ; issue EOI to master 8259 interrupt
        out     20h, al                         ; controller
        out     0A0h, al                ; issue EOI to slave 8259
        xor     ax, ax                  ; Clear 387 BUSY signal
        out     0F0h, al                        ;

;
; Get the coprocessor environment and print it out
;
        mov     ax, SS_DATA             ; set DS to our data segment
        mov     ds, ax                          ;
        fstenv  envblk                  ; get 387 environment
        call    display_env             ; displays coprocessor environment
```

```
;
; Clear the 387 status and return to interrupted code.
;
        wait                        ; clear exception bits in 387 status
        fclex                       ;
        fldcw   ctl_387             ; unmask all 387 exceptions
        pop     ds                  ; restore regs
        pop     eax                 ;
        iretd                       ; return to interrupted code
irq13_hndlr endp
```

The first thing the handler does is re-enable interrupts and clear the source of the exception. This allows other hardware interrupts to occur. It also allows IRQ13 to occur again, so you should not acknowledge the interrupt until you get to a point in your handler where it's OK if the handler gets reentered. In the case of the coprocessor exception, we know it won't occur again because the handler isn't going to do anything to cause another 387 exception, so we can acknowledge the interrupt immediately.

The handler then gets the 387 environment and prints it on the display, clears the 387 flag that says the exception occurred, and returns to the interrupted code.

CTRLC.ASM

The CTRLC.ASM example program installs a protected mode CTRL-C handler that calls the C library printf() routine to print a string. This program demonstrates how to correctly call high level language code from within an interrupt handler.

The CTRLC program prompts the user to type CTRL-C; the handler then prints a string on the display, and the program terminates. The program output looks like:

```
Please type CTRL-C
^C
CTRL-C was typed
```

The INT 23h interrupt handler from CTRLC.ASM follows:

```
;*************************************************************************
; CC_HNDLR - Protected Mode CTRL-C Handler
;       This handler calls through to the C printf() routine to print
;       a string saying that CTRL-C occurred, and sets a global flag so
;       the main loop knows that CTRL-C has occurred.
;
;       In order to call a C routine, we must first set up a stack that
;       is in segment 0014h, and also set DS and ES to segment 0014h.
;*************************************************************************

        extrn   printf:near             ; C library routine

        public  cc_hndlr
cc_hndlr        proc    near
;
; This is an interrupt handler, so we must preserve all registers that
; may get modified by us or the C routine we will call.  Also re-enable
; interrupts so hardware interrupts aren't blocked.
;
        sti                             ; re-enable interrupts
        pushad                          ; save all registers
        push    ds                              ;
        push    es                              ;
        push    fs                              ;
        push    gs                              ;

;
; Set up a C environment, including a stack in our data segment, and call
; the C printf() library routine to print a string.
;
; Note the SS selector must have its RPL bits set to the current privilege
; level (unlike DS and ES, where we can just use a hardwired RPL of zero).
;
        mov     ax,SS_DATA              ; set DS and ES to our data seg
        mov     ds,ax                           ;
        mov     es,ax                           ;
        mov     ecx,esp                 ; get current stack pointer
        mov     dx,ss                           ;
        mov     ax,pgm_ss               ; switch to stack in data segment
        mov     ss,ax                           ;
        lea     esp,cc_tos                      ;
        push    edx                     ; save old stack pointer on new stack
        push    ecx                             ;
        lea     eax,cc_msg              ; call printf to print string
        push    eax                             ;
        call    printf                          ;
        add     esp,4                           ;
        lss     esp,pword ptr [esp]     ; switch back to old stack
```

```
;
; Set a flag so the main loop knows the CTRL-C occurred, restore registers,
; and return to caller
;
        mov     cc_flag,TRUE            ; CTRL-C has been processed
        pop     gs                      ; restore regs
        pop     fs                              ;
        pop     es                              ;
        pop     ds                              ;
        popad                                   ;
        iretd                           ; return from interrupt
cc_hndlr        endp
```

To call code written in a high level language from an interrupt handler, preserve all registers that may get modified (depending on the register saving rules of the language you are using), and set up the environment expected by the compiler. For most high level languages, this means DS and ES must point to the program data segment (selector 0014h), and the current stack must also be in the program data segment. For the SS register, the RPL (requested privilege level) bits must always be at the current privilege level, so we load SS with a selector value we saved at init time. For DS, ES, FS, and GS, it's always OK to use a hardwired RPL value of zero when loading the segment register.

The handler shown above uses a buffer allocated in the program data segment as a stack when calling C code. This handler is not reentrant as shown, because it always uses the same data buffer for the stack.

SERIO.ASM

The SERIO.ASM example program installs bimodal (separate real-mode and protected-mode) interrupt handlers for the IRQ4 (COM1 serial port) hardware interrupt. The interrupt handlers store characters received over the serial line into a circular buffer. The main program loop retrieves characters out of the buffer and prints them on the screen. When a carriage return is received, the program terminates.

The program does **not** set up the serial port parameters; use the MS-DOS MODE command to program the port appropriately before running this program. For example, to use 9600 baud, no parity, 8 data bits, one stop bit, use the command:

```
mode com1 9600,n,8,1
```

To test this rogram, start it running on an 80386 PC after initializing the port parameters. From another PC, use a communications program to send characters to the COM1 serial port on the first PC. Send an ASCII carriage return to terminate the program.

The entire SERIO.ASM program follows:

```
;
; Constants and data structures
;
include dosx.ah

IOBUF_SIZE      equ     32                      ; Size of circular buffer
CR              equ     0Dh                     ; ASCII carriage return
LF              equ     0Ah                     ; ASCII linefeed
PIC_MSK         equ     21h                     ; PIC Mask Register port #
PIC_EOI         equ     20h                     ; PIC EOI Register port #
PORTNO          equ     3F8h                    ; COM1 base address (port #)
COM_IER         equ     PORTNO+1                ; Serial port INT Enable Reg
COM_MCR         equ     PORTNO+4                ; Serial port Modem Ctrl Reg
IRQ4_MSK        equ     10h                     ; PIC enable IRQ4 INTs mask

;
; Struct for data kept in conventional memory that also needs to be accessed
; from protected mode
;
REALDATA struc
        RD_IDX          dd      ?       ; current read index in IO_BUF
        WR_IDX          dd      ?       ; current write index in IO_BUF
        IO_BUF  db IOBUF_SIZE dup (?)   ; 32 byte circular I/O buffer
REALDATA ends
```

```
;
; Segment definitions and ordering
;
_rcodeseg       segment byte public use16 'rm_code'
        public  start_real
start_real      label   byte
_rcodeseg       ends
_rdata          segment dword public use16 'rm_data'
_rdata          ends
_data           segment dword public use32 'data'
_data           ends
dgroup  group   _rdata, _data
_codeseg        segment byte public use32 'code'
_codeseg        ends
_stack          segment dword stack use32 'stack'
        db      2048 dup (?)    ; 2K stack
_stack          ends


;
; Global data that needs to be accessed in both real mode and protected mode
; This data gets copied to conventional memory;  it is not used in its
; original link-time location.
;
_rdata  segment

io_data REALDATA        <>      ; I/O buffer and circular buffer pointers

        public  end_real
end_real        label   byte

_rdata  ends

;
; Global protected mode data
;
_data   segment

;
; Original RM and PM IRQ4 vectors
;
        public  rm_irq4_vec,pm_irq4_off,pm_irq4_sel,irq4_int
rm_irq4_vec     dd      ?       ; original real mode vector
pm_irq4_off     dd      ?       ; original prot mode vector
pm_irq4_sel     dw      ?               ;
irq4_int        db      ?       ; interrupt number for IRQ4
        align   4
```

```
;
; Conventional memory block control.  Note that the real mode CS and DS
; values for our real mode IRQ4 handler are not necessarily the same as
; the address of the memory block.  This is because if necessary we back
; them off so the link-time offsets to code and data will still be correct
; at run time.  It is necessary to do this if the real mode code and data
; don't begin at the start of the protected mode segment (eg, if the
; -OFFSET 1000h link-time switch is used).
;
cbuf_seg        dw      ?       ; real mode paragraph addr of memory block
rm_csds         dw      ?       ; real mode CS and DS values
        align   4
iodat_offs      dd      ?       ; offset in segment 0034h (the MS-DOS memory
                                        ; seg) of the io_data structure

prompt_msg db   'Send CR-terminated data over serial port 1',0Dh,0Ah,'$'
        align   4

_data   ends

;****************************************************************************
; Program entry point
;****************************************************************************

        assume  cs:_codeseg,ds:_data
_codeseg        segment

        public  main
main proc       near

;
; Copy the real mode code and data to a conventional memory block, and init
; globals for conv mem block
;
        call    copy_real               ; set up real mode code/data
        cmp     eax,TRUE                ; exit if error
        je      #exit                           ;
```

```
;
; Save current real and prot mode IRQ4 vectors
;
        mov     ax,250Ch                ; get IRQ4 interrupt number
        int     21h                             ;
        add     al,4                            ;
        mov     irq4_int,al                     ;
        mov     cl, irq4_int            ; save real mode IRQ4 vector
        mov     ax, 2503h                       ;
        int     21h                             ;
        mov     rm_irq4_vec, ebx                ;
        mov     ax, 2502h               ; save prot mode IRQ4 vector
        mov     cl,irq4_int                     ;
        int     21h                             ;
        mov     pm_irq4_sel, es                 ;
        mov     pm_irq4_off, ebx                ;

;
; Install our real-mode and protected-mode IRQ4 (COM1) handlers.
;
        push    ds                      ; set real and prot mode IRQ4 vectors
        mov     cl,irq4_int                     ;
        mov     bx,rm_csds                      ;
        shl     ebx,16                          ;
        lea     bx,rm_irq4_hnd                  ;
        lea     edx,pm_irq4_hnd                 ;
        mov     ax,cs                           ;
        mov     ds,ax                           ;
        mov     ax,2507h                        ;
        int     21h                             ;
        pop     ds                              ;

;
; Init MCR and IER in the 8250 serial chip.
; Make sure IRQ4 is unmasked in the 8259 interrupt controller.
;
        mov     dx, COM_MCR             ; set RTS, DTR, and OUT2 bits in the
        mov     al, 0Bh                         ; Modem Control Register
        out     dx, al                          ;
        mov     dx, COM_IER             ; enable interrupts on receive data
        mov     al, 01h                         ; on the serial chip
        out     dx, al                          ;
        in      al, PIC_MSK             ; unmask IRQ4 in the 8259 mask
        and     al, NOT IRQ4_MSK                ;
        out     PIC_MSK, al                     ;

;
; Prompt user to send serial data
;
        mov     ah, 09h                 ; print prompt message
        lea     edx, prompt_msg                 ;
        int     21h                             ;
```

```
;
; Loop, reading data from the buffer as it gets filled and printing
; it to the screen.  When we see a carriage return, exit the loop.
;
        mov     ax,SS_DOSMEM            ; set ES:EBX to I/O data block in
        mov     es,ax                           ; conventional memory
        mov     ebx,iodat_offs                  ;
#loop:
        mov     ecx,es:[ebx].RD_IDX     ; keep looping if buffer empty
        cmp     ecx,es:[ebx].WR_IDX             ;
        je      #loop                           ;
        pushfd                          ; get character and update read index,
        cli                                     ; disabling interrupts while
        mov     dl,es:[ebx][ecx].IO_BUF         ; we update the I/O globals
        inc     ecx                             ;
        cmp     ecx,IOBUF_SIZE                  ;
        jb      short #in_buf                   ;
        mov     ecx,0                           ;
#in_buf:                                        ;
        mov     es:[ebx].RD_IDX,ecx             ;
        popfd                                   ;
        mov     ah, 02h                 ; print the character in DL on the
        int 21h                                 ; display
        cmp     dl,CR                   ; continue loop unless it was CR
        jne     #loop                           ;
        mov     dl,LF                   ; print a line feed
        mov     ah,2                            ;
        int     21h                             ;
```

```
;
; Restore serial port to previous state, restore IRQ4 handlers, and
; free up the conventional memory block we allocated.
;
        mov     dx, COM_MCR             ; clear RTS, DTR, and OUT2 in the
        in      al, dx                          ; Modem Control Register
        and     al, 04h                         ;
        out     dx, al                          ;
        mov     dx, COM_IER             ; disable serial chip interrupts
        mov     al, 00h                         ;
        out     dx, al                          ;
        in      al, PIC_MSK             ; mask off IRQ4 in the 8259
        or      al, IRQ4_MSK                    ;
        out     PIC_MSK, al                     ;
        push    ds                      ; restore original real and prot mode
        mov     cl,irq4_int                     ; IRQ4 vectors
        mov     ebx, rm_irq4_vec                ;
        mov     edx, pm_irq4_off                ;
        mov     ax, pm_irq4_sel                 ;
        mov     ds, ax                          ;
        mov     ax,2507h                        ;
        int     21h                             ;
        pop     ds                              ;
        mov     cx,dbuf_seg             ; free up conventional memory block
        mov     ax,25C1h                        ;
        int     21h                             ;

#exit:
        mov     ax, 4C00h               ; exit to MS-DOS
        int     21h                             ;
main endp
```

```
;************************************************************************
; COPY_REAL — copy real mode code & data to a conventional memory block,
;       and initialize global data used to manage the memory block
;
; Returns:  TRUE        if error
;           FALSE       if success
;************************************************************************
        public  copy_real
copy_real proc  near
;
; Stack frame
;
#REAL_NBYTES equ (dword ptr 12[ebp])    ; number of bytes RM code & data
#REAL_START equ (dword ptr 8[ebp])      ; start offset of RM code & data

        push    ebp                     ; set up stack frame
        mov     ebp,esp                 ;
        sub     esp,8                   ; allocate local data
        push    es                      ; save regs
        push    esi                     ;
        push    edi                     ;

;
; Allocate MS-DOS segment in conventional memory the size of our real mode
; code and data.
;
; NOTE we are just assuming there is free conventional memory (the program
;       should have been linked with the -MINREAL switch), rather than
;       attempting to make sure memory is available with the 2525h and/or
;       2530h system calls as we would in a more elaborate program.
;
        lea     ebx, end_real           ; end of RM code and data
        test    ebx,0FFFF0000h          ; branch if not within 1st 64K of
        jnz     #bad_segorder                   ; of program segment
        lea     ecx, start_real         ; start of RM code and data, rounded
        and     ecx, not 0Fh                    ; down to paragraph boundary
        mov     #REAL_START, ecx                ;
        sub     ebx, ecx                ; RM code and data size in EBX
        mov     #REAL_NBYTES, ebx               ;
        add     ebx, 15                 ; round size up to paragraph count
        shr     ebx, 4                          ;
        mov     ax, 2500h               ; alloc MS-DOS memory block
        int     21h                             ;
        jc      #no_mem                 ; branch if failure
        mov     dbuf_seg, ax            ; save addr of conv mem block
```

```
;
; Calculate a real mode segment address for real mode CS and DS that will
; allow the real mode code to use all its link-time offsets to code and data.
; Copy the real mode code and data to conventional memory.
;
        mov     ax,cbuf_seg             ; calculate real-mode CS and DS
        mov     ebx,#REAL_START                 ;
        shr     ebx,4                           ;
        sub     ax,bx                           ;
        mov     rm_csds,ax                      ;
        push    ds                      ; copy code & data to conv mem block
        mov     ecx, #REAL_NBYTES               ;
        mov     ax, SS_DOSMEM                   ;
        mov     es, ax                          ;
        movzx   edi, cbuf_seg                   ;
        shl     edi, 4                          ;
        mov     esi, #REAL_START                ;
        mov     ax, cs                          ;
        mov     ds, ax                          ;
        cld                                     ;
        rep     movsb                           ;
        pop     ds                              ;

;
; Calculate the protected mode offset in segment 0034h to the I/O data
; block we keep in conventional memory.  Also initialize the data block.
;
        lea     eax,io_data             ; get prot mode offset to I/O data
        movzx   ebx,rm_csds                     ; in conv memory block
        shl     ebx,4                           ;
        add     eax,ebx                         ;
        mov     iodat_offs,eax                  ;
        mov     ax,SS_DOSMEM            ; init to empty buffer
        mov     es,ax                           ;
        mov     eax,iodat_offs                  ;
        mov     es:[eax].RD_IDX,0               ;
        mov     es:[eax].WR_IDX,0               ;

        mov     eax,FALSE               ; return success
#exit:
        pop     edi                     ; restore regs
        pop     esi                             ;
        pop     es                              ;
        mov     esp,ebp                 ; restore stack frame & exit
        pop     ebp                             ;
        ret                                     ;

#no_mem:
```

```
;
; Couldn't allocate conventional memory for real mode code & data
;
_data   segment
nocmem_msg db   "Can't allocate conventional memory buffer",0Dh,0Ah,'$'
_data   ends
        mov     ah, 09h                 ; print err message
        lea     edx, nocmem_msg                 ;
        int     21h                             ;
        mov     eax,TRUE                ; return error
        jmp     #exit                           ;

#bad_segorder:
;
; real mode code & data not within 1st 64K of program segment
;
_data   segment
boffs_msg db    'Real mode code/data > 64K into program segment',0Dh,0Ah,'$'
_data   ends
        mov     ah, 09h                 ; print err message
        lea     edx, boffs_msg                  ;
        int     21h                             ;
        mov     eax,TRUE                ; return error
        jmp     #exit                           ;
copy_real endp

;****************************************************************************
; PM_IRQ4_HND - Protected Mode IRQ4 Interrupt Handler
;       Just reads the character from the serial port and stuffs it into
;       the circular buffer.
;****************************************************************************
        public  pm_irq4_hnd
pm_irq4_hnd     proc    near

        push    eax                     ; save regs we use
        push    ebx                             ;
        push    ecx                             ;
        push    edx                             ;
        push    es                              ;
        push    ds                              ;
        mov     ax,SS_DATA              ; set DS to our data segment
        mov     ds, ax                          ;
        mov     ax,SS_DOSMEM            ; set ES:EBX to I/O data in
        mov     es, ax                          ; conventional memory
        mov     ebx,iodat_offs                  ;
```

```
;
; Get the character and write it to the buffer, with interrupts still
; disabled so we can't get reentered.
;
; If the circular buffer is full, just drop this character on the floor.
;
        mov     dx, PORTNO              ; read character from serial chip
        in      al, dx                          ;
        mov     ecx,es:[ebx].WR_IDX     ; calculate new write index after
        inc     ecx                             ; we stuff this char into
        cmp     ecx,IOBUF_SIZE                  ; buffer
        jb      short #in_buf                   ;
        mov     ecx,0                           ;
#in_buf:                                        ;
        cmp     ecx,es:[ebx].RD_IDX     ; drop this char if buffer full
        je      short #done_ch                  ;
        mov     edx,es:[ebx].WR_IDX     ; write this char to buffer
        mov     es:[ebx][edx].IO_BUF,al         ;
        mov     es:[ebx].WR_IDX,ecx     ; update write index
#done_ch:

;
; It's now OK to get reentered, so we can re-enable interrupts and
; acknowledge the interrupt to the 8259
;
        sti                             ; re-enable interrupts
        mov     al, 20h                 ; send EOI to 8259
        out     PIC_EOI, al                     ;

        pop     ds                      ; restore registers
        pop     es                              ;
        pop     edx                             ;
        pop     ecx                             ;
        pop     ebx                             ;
        pop     eax                             ;
        iretd                           ; return to interrupted code

pm_irq4_hnd     endp

_codeseg        ends
```

```
;*****************************************************************************
; RM_IRQ4_HND - Real Mode IRQ4 Interrupt Handler
;       Just reads the character from the serial port and stuffs it into
;       the circular buffer.
;*****************************************************************************
        assume  cs:_rcodeseg, ds:dgroup
_rcodeseg segment

        public  rm_irq4_hnd
rm_irq4_hnd     proc    near

        push    eax                     ; save regs we use
        push    ebx                             ;
        push    ecx                             ;
        push    edx                             ;
        push    ds                              ;
        mov     ax,cs                   ; set DS to our data segment
        mov     ds,ax                           ;

;
; Get the character and write it to the buffer, with interrupts still
; disabled so we can't get reentered.
;
; If the circular buffer is full, just drop this character on the floor.
;
        mov     dx, PORTNO              ; read character from serial chip
        in      al, dx                          ;
        mov     ecx,io_data.WR_IDX      ; calculate new write index after
        inc     ecx                             ; we stuff this char into
        cmp     ecx,IOBUF_SIZE                  ; buffer
        jb      short #in_buf                   ;
        mov     ecx,0                           ;
#in_buf:                                        ;
        cmp     ecx,io_data.RD_IDX      ; drop this char if buffer full
        je      short #done_ch                  ;
        mov     edx,io_data.WR_IDX      ; write this char to buffer
        mov     io_data[edx].IO_BUF,al          ;
        mov     io_data.WR_IDX,ecx      ; update write index
#done_ch:

;
; It's now OK to get reentered, so we can re-enable interrupts and
; acknowledge the interrupt to the 8259
;
        sti                             ; re-enable interrupts
        mov     al, 20h                 ; send EOI to 8259
        out     PIC_EOI, al                     ;

        pop     ds                      ; restore registers
        pop     edx                             ;
        pop     ecx                             ;
        pop     ebx                             ;
```

```
            pop     eax                                 ;
            iret                                ; return to interrupted code

rm_irq4_hnd     endp

_rcodeseg ends

            end main
```

The interrupt handlers for the SERIO program are relatively simple. The interesting feature of this program is its use of both a real mode and protected mode interrupt handler to avoid the overhead of a mode switch when a hardware interrupt occurs. This is an appropriate technique for high frequency interrupts.

The initialization routine (COPY_REAL) allocates a conventional memory block and copies the real mode code and data for the real mode IRQ4 handler to the memory block. The program is organized so that the real mode code and data is placed at the beginning of the program segment. By keeping the real mode code and data within the first 64KB of the protected mode program, the real mode segment can be set up so link-time offsets to data are correct.

The -REALBREAK switch is not used to automatically load the real mode code and data in conventional memory because -REALBREAK is not DPMI-compatible.

CRITERR.ASM

The CRITERR.ASM example program installs an INT 24h (MS-DOS critical error) handler, and then causes a critical error by attempting to open a file on the A: floppy disk drive (which causes a critical error if the floppy door is open). The critical error handler prints an error message and returns, instructing MS-DOS to fail the function call. The handler retrieves the original AH function code (saved on the real mode stack by MS-DOS before issuing the INT 24h) from the INT 21h and prints it, to demonstrate how to use the protected mode interrupt stack frame to look at segment registers and the original stack at the time the interrupt occurred.

The CRITERR program prints the following output:

```
Causing critical err by reading from open A: floppy drive
Critical error occurred on INT 21h function 3D
```

The protected mode critical error interrupt handler follows:

```
;****************************************************************************
; CRITE_HND - Protected Mode Critical Error Handler
;       It just prints out a message saying the error occurred, and then
;       returns, instructing MS-DOS to fail the system call that caused the error.
;
;       To demonstrate how to use the interrupt stack frame to get the
;       original stack when the interrupt occurred, we pick up the
;       original AX value for the INT 21h call (saved on the critical error
;       stack by MS-DOS) and print out AH to show what call it was that failed.
;****************************************************************************
        public  crite_hnd
crite_hnd       proc    near
;
; Interrupt handler stack frame
;
#FLGS   equ     (dword ptr 52[ebp])     ; 386|DOS-Extender flags
#GS     equ     (word ptr 48[ebp])      ; original GS
#FS     equ     (word ptr 44[ebp])      ; original FS
#DS     equ     (word ptr 40[ebp])      ; original DS
#ES     equ     (word ptr 36[ebp])      ; original ES
#SS     equ     (word ptr 32[ebp])      ; original SS
#ESP    equ     (dword ptr 28[ebp])     ; original ESP
#EFLAGS equ     (dword ptr 24[ebp])     ; original EFLAGS
#CS     equ     (dword ptr 20[ebp])     ; original CS
#EIP    equ     (dword ptr 16[ebp])     ; original EIP
#EBP    equ     (dword ptr [ebp])       ; original EBP

;
; This is an interrupt handler, so we must preserve all registers we use.
; Also re-enable interrupts so hardware interrupts aren't blocked.
;
        sti                             ; re-enable interrupts
        push    ebp                     ; set up stack frame
        mov     ebp,esp                         ;
        push    ebx                     ; save regs, EXCEPT EAX which we
        push    ecx                             ; return a value in
        push    edx                             ;
        push    ds                              ;
        push    es                              ;
        mov     ax,SS_DATA              ; set DS to our data segment
        mov     ds,ax                           ;
```

```
;
; Get the real mode stack address in ES:EBX.  The critical error
; interrupt should ALWAYS be issued from real mode, but check in case
; someone coded a software INT 24h in prot mode by mistake.
;
        test    #FLGS,IFL_RMODE         ; branch if didn't come from real
        jz      #not_real                       ; mode
        movzx   ebx,#SS                 ; get linear addr of real mode stack
        shl     ebx,4                           ; in conventional memory
        add     ebx,#ESP                        ;
        mov     ax,SS_DOSMEM            ; set ES to MS-DOS memory segment
        mov     es,ax                           ;

;
; Print out a message, and the AH value for the INT 21h call that failed,
; so the user can see what the call was.
;
; MS-DOS critical error stack frame
;
#DOS_FLAGS equ  (word ptr es:[ebx])     ; interrupt stack frame from real
#DOS_CS equ     (word ptr es:[ebx])             ; mode INT 21h
#DOS_IP equ     (word ptr es:[ebx])             ;
#DOS_ES equ     (word ptr es:[ebx])     ; regs at time INT 21h was issued
#DOS_DS equ     (word ptr es:[ebx])             ; in real mode
#DOS_BP equ     (word ptr es:[ebx])             ;
#DOS_DI equ     (word ptr es:[ebx])             ;
#DOS_SI equ     (word ptr es:[ebx])             ;
#DOS_DX equ     (word ptr es:[ebx])             ;
#DOS_CX equ     (word ptr es:[ebx])             ;
#DOS_BX equ     (word ptr es:[ebx])             ;
#DOS_AX equ     (word ptr es:[ebx])             ;

_data   segment
ce_msg  db      'Critical error occurred on INT 21h function $'
_data   ends
        mov     ah,9                    ; print critical err msg
        lea     edx,ce_msg                      ;
        int     21h                             ;
        mov     ax,#DOS_AX              ; get original AX value
        mov     al,ah                   ; convert AH to ASCII, save in BX
        call    btohex                          ;
        mov     bx,ax                           ;
        mov     ah,2                    ; output function code & newline
        mov     dl,bl                           ;
        int     21h                             ;
        mov     dl,bh                           ;
        int     21h                             ;
        mov     dl,0Dh                          ;
        int     21h                             ;
        mov     dl,0Ah                          ;
        int     21h                             ;
```

```
#exit:
        mov     eax,3                   ; instruct MS-DOS to fail the function
                                                ; (MS-DOS 3.1 and later only)
        pop     es                      ; restore regs
        pop     ds                              ;
        pop     edx                             ;
        pop     ecx                             ;
        pop     ebx                             ;
        pop     ebp                             ;
        iretd                           ; return from interrupt

#not_real:
;
; Interrupt didn't originate from real mode
;
_data   segment
nreal_msg db    'Critical error interrupt issued in prot mode!!',0Dh,0Ah,'$'
_data   ends
        mov     ah,9                    ; print error msg & exit
        lea     edx,nreal_msg                   ;
        int     21h                             ;
        jmp     #exit                           ;

crite_hnd       endp
```

The handler first looks at the protected mode interrupt stack to determine what the SS:ESP was at the time the INT 24h was issued. In this case, since the critical error interrupt is issued in real mode by MS-DOS, it's a real mode selector in SS, so we have to turn the real mode SS:SP into an offset in the first megabyte and use segment 0034h to get to it.

MS-DOS saves all the registers at the time of the INT 21h that caused the critical error on the real mode stack. So we retrieve the original AX from the real mode stack, and print out the AH value (3Dh, for the file open call that was made) from the original INT 21h.

Writing DPMI-Compatible Applications

The DOS Protected Mode Interface (DPMI) is an interface that is provided by a multitasking environment such as Windows, and used by 386 | DOS-Extender to run in that environment. DPMI was introduced in 1990 in version 3.0 of Microsoft Windows. Future versions of OS/2 and UNIX DOS boxes will likely provide DPMI.

Although version 3.0 of 386 | DOS-Extender does not support DPMI, the next major release will support DPMI. The lack of DPMI support means that 386 | DOS-Extender version 3.0 cannot run in Windows enhanced mode; however, 386 | DOS-Extender can run in the Windows standard and real modes of operation.

You may want to plan for DPMI compatibility in your application. There are several points to consider:

- Under DPMI, applications are always unprivileged (run at a privilege level other than zero). Make sure your program can run with the -UNPRIVILEGED switch set, and don't assume privilege level three (the privilege level is determined by the multitasking host; under Windows 3.0 it is level one). See sections 3.1 and 4.1 for more information on privilege levels.
- Some 386|DOS-Extender switches cannot be supported under DPMI. If possible, restrict your application to DPMI-compatible switches.

☛ Some 386|DOS-Extender system calls cannot be supported under DPMI. If possible, restrict your application to DPMI-compatible system calls, or be prepared for system calls that normally succeed to return an error in a DPMI environment.

☛ 386|VMM is not supported under DPMI. Applications bound to 386|VMM are run without 386|VMM under a DPMI host. Since most DPMI hosts provide their own virtual memory support, the application can still use large amounts of virtual memory. The 386|VMM-specific switches and system calls behave as they always do when 386|VMM is not present: the switches are ignored, and the system calls return success without doing anything. This behavior allows programs that have been built to run with 386|VMM to run successfully when 386|VMM is not present.

F.1 DPMI Versions and Capabilities

There are two versions of DPMI. Version 0.90 is implemented in Windows 3.0. Version 1.00 is defined but not yet implemented. More 386 | DOS-Extender functionality can be supported under DPMI 1.00 than under DPMI 0.90. In addition, there are some optional features in DPMI 1.00 that may or may not be supported in any specific DPMI environment.

To determine which version you have, use the Get Configuration Information system call (function 2526h) to detect a DPMI environment, obtain the DPMI interface version number, and determine which DPMI capabilities are supported.

Then, use the following rules to check for 386 | DOS-Extender features optionally supported under DPMI:

☛ If DPMI is present, initially assume that only those 386|DOS-Extender features that are unconditionally supported under DPMI are available.

☛ Check the DPMI capabilities flags returned by the 2526h system call to see which features optionally supported under DPMI are available.

The names and meaning of the optional DPMI capabilities are described below; none of them are supported under DPMI 0.90. These are the DPMI optional capabilities:

- Paging Support: This capability allows 386|DOS-Extender to provide unmapped pages within the main program segment. Without this capability, the -OFFSET switch for null pointer detection cannot be supported. The Paging Support capability is always supported under DPMI 1.00.
- Device Mapping: This capability allows 386|DOS-Extender to support mapped physical device pages within the main program segment. Without this capability, physical devices (such as the video memory and the Weitek coprocessor) must be mapped in separate segments and accessed with FAR pointers. Even when this capability is present, the DPMI host can choose to fail individual mapping attempts; for example, hosts with this capability would likely support mapping of the Weitek chip in the main program segment, but not mapping of the video memory. The Device Mapping capability is optionally supported under DPMI 1.00.
- Conventional Memory: This capability allows 386|DOS-Extender to support the -REALBREAK switch. The Conventional Memory capability is optionally supported under DPMI 1.00.
- Exceptions Restartability: This capability means exceptions within the DPMI host kernel can be restarted. No 386|DOS-Extender functionality relies on this DPMI capability. The Exceptions Restartability capability is optionally supported under DPMI 1.00.
- Page Accessed/Dirty: This capability means the DPMI host makes the page dirty and accessed bits in page table entries available. No 386|DOS-Extender functionality relies on this DPMI capability. The Page Accessed/Dirty capability is optionally supported under DPMI 1.00.

F.2 Interrupt Handlers and Interrupt Flag Control under DPMI

Interrupt and exception handlers see the same interrupt stack frame under DPMI as in any other environment. There are two special considerations when writing interrupt handlers to be DPMI compatible:

- The interrupt enable flag (IF) in the EFLAGS register should not be managed with the PUSHFD/POPFD instructions, and is only set correctly by an IRETD when returning from an interrupt handler (not when using an IRETD as a convenient way to load a new CS:EIP and EFLAGS). See section 6.7 for rules for DPMI-compatible management of the interrupt flag.

- ☛ Interrupt handlers that retain control and do not return to the interrupted code must allow the DPMI host to remove the interrupt context. The handler must modify the return CS:EIP and SS:ESP to point to its own code and stack, and execute an IRETD to allow the DPMI host to clean up. See section 6.6.5 for more information on retaining control in an interrupt handler.

F.3 386 | DOS-Extender Switches

Most 386 | DOS-Extender switches not supported under DPMI are ignored in a DPMI environment. However, there are a few switches (such as -REALBREAK and -OFFSET) which cannot be ignored; 386 | DOS-Extender prints an error message and refuses to run the program if these switches are used under a DPMI host. The table below lists all switches that are unsupported or optionally supported under DPMI; all other 386 | DOS-Extender switches are unconditionally available.

System Call	Supported if DPMI capability present	Comment
386 | VMM switches (see section 2.11)	no	switches are ignored
-EXTHIGH	no	switch is ignored
-EXTLOW	no	switch is ignored
-HWIVEC	no	switch is ignored
-INTMAP	no	switch is ignored
-NOPAGE	no	application program cannot run
-OFFSET	Paging Support	application program cannot run unless Paging Support capability is present
-REALBREAK	Conventional Memory	application program cannot run unless Conventional Memory capability is present
-PRIVILEGED	no	application program cannot run

F.4 386 | DOS-Extender System Calls

Most 386 | DOS-Extender system calls not supported under DPMI return error code 130 (not available under DPMI). The 386 | VMM-specific calls behave as documented without 386 | VMM (usually returning success without doing anything). The table below lists all system calls that are unsupported, partially supported, or optionally supported under DPMI; all other 386 | DOS-Extender system calls are unconditionally available.

System Call	Supported if DPMI capability present	Comment
2509h	no	Convert Linear Addr to Physical Addr: always returns error (carry flag set).
250Ah, nonzero segments	Device Mapping	Map Physical Memory at End of Segment: always supported for zero-sized segments. For nonzero segments, returns error code 130 if Device Mapping capability not present.
250Fh	Conventional Memory	Convert Prot Mode Addr to MS-DOS Addr: returns error (carry flag set) if Conventional Memory capability not present.
2513h, system segments	no	Alias Segment Descriptor: only code and data segments can be created with this call under DPMI.
251Dh	no	Read Page Table Entry: always returns error code 130.
251Eh	no	Write Page Table Entry: always returns error code 130.
251Fh	no	Exchange Page Table Entries: always returns error code 130.
2523h	no	Specify Out-of-Swap-Space Handler: always returns success but does nothing.

System Call	Supported if DPMI capability present	Comment
2524h	no	Specify Page Replacement Handlers: always returns success but does nothing.
252Bh, BH=0	Paging Support	Create Unmapped Pages: returns error code 130 if Paging Support capability not present.
252Bh, BH=1	Paging Support	Create Allocated Pages: returns error code 130 if Paging Support capability not present.
252Bh, BH=2	Device Mapping	Create Physical Device Pages: returns error code 130 if Device Mapping capability not present.
252Bh, BH=3	no	Map Data File: always returns error code 129 (invalid parameter, because 386 I VMM not present).
252Ch	Paging Support	Add Unmapped Pages to Segment: returns error code 130 if Paging Support capability not present.
252Dh	no	Close 386 I VMM File Handle: always returns error code 129 (invalid file handle), because since 386 I VMM is not present there is no way to get a valid 386 I VMM file handle.
252Eh	no	Get/Set 386 I VMM Parameters: always returns success but does nothing; the caller's buffer is zeroed when getting parameters.
252Fh	no	Write Record to 386 I VMM Page Log File: always returns error code 133 (no page log file), because 386 I VMM is not present.
253Ch	no	Shrink 386 I VMM Swap File: always returns success but does nothing.

Hardware Interrupts under VCPI and DESQview

The Virtual Control Program Interface (VCPI) supports reprogramming the 8259 interrupt controllers to relocate hardware interrupts from their standard MS-DOS interrupt vectors of 08h-0Fh for IRQ0-7 and 70h-77h for IRQ8-15. In many cases (e.g., under most EMS emulators) the interrupt controllers are not reprogrammed and hardware interrupts actually do occur on the normal vectors. More elaborate environments such as DESQview 386 do relocate hardware interrupts.

By default, 386 | DOS-Extender always presents hardware interrupts to the application at the standard interrupt vector locations, regardless of whether the VCPI host has reprogrammed the interrupt controllers. Applications therefore need not (and should not) be sensitive to whether VCPI is present.

Although not recommended, for backward compatibility 386 | DOS-Extender does support the ability to see hardware interrupts at their actual hardware vector mappings under VCPI. The remainder of this appendix describes in more detail how hardware interrupts work in a VCPI environment, and how the application program can use the -HWIVEC switch to modify the 386 | DOS-Extender hardware interrupt processing under VCPI.

G.1 How Hardware Interrupts Work under VCPI

Under VCPI, real mode code (including MS-DOS and the BIOS) runs in the Virtual 8086 (V86) mode of the 80386 processor. 386 | DOS-Extender switches to V86 mode rather than real mode to reissue DOS system calls and other interrupts that occur while the application program is running in protected mode.

When the application is running in protected mode, 386 | DOS-Extender has control of the machine. If a hardware interrupt occurs, 386 | DOS-Extender sees it directly on the interrupt vector that the 8259 is programmed for.

When real mode code is running in V86 mode, the VCPI host has control of the machine. If a hardware interrupt occurs, the VCPI host gets control in its own, separate, protected mode environment. Usually, it just reflects all interrupts, including hardware interrupts, back to the real mode interrupt handler in V86 mode. If, however, hardware interrupts are relocated, it reflects them to the handler at the standard MS-DOS interrupt vector rather than the relocated vector. For example, suppose DESQview has reprogrammed the interrupt controllers such that IRQ0-7 interrupt on vectors 50h-57h. When a timer tick (IRQ0) occurs in V86 mode, DESQview gets control in its protected mode INT 50h handler. But instead of reflecting the interrupt to the real mode INT 50h handler, DESQview reflects it to the real mode INT 08h handler, the standard interrupt vector for IRQ0.

So if 386 | DOS-Extender did nothing special, the application would see IRQ0 occurring on different vectors in protected mode and real mode for the above example. In protected mode, it would have to install an INT 50h handler to trap timer ticks, and in real mode it would install an INT 08h handler.

It's too complicated to deal with different vectors for real mode and protected mode. So by default, 386 | DOS-Extender makes it appear to the application as though the hardware interrupts occur in their standard locations in protected mode as well as real mode. When the application sets a protected mode hardware interrupt vector, 386 | DOS-Extender automatically sets the actual hardware interrupt vector as well as the standard MS-DOS interrupt vector. In our example, if the application installs a protected mode INT 08h handler with system call 2506h (always get control in protected mode), the call sets the real mode INT 08h vector, and the protected mode INT 08h and INT 50h vectors. If the interrupt occurs in V86 mode, it gets passed up to protected mode and reissued, and the application handler gets control through INT 08h. If the interrupt occurs in protected mode, the application handler is invoked through INT 50h. The only difference the handler sees is the interrupt number in the stack frame.

The 250Ch call (Get Hardware Interrupt Vectors) by default always returns the standard MS-DOS hardware vectors (08h and 70h). The 2526h call (Get Configuration Information) returns the actual hardware interrupt vectors under VCPI.

G.2 The -HWIVEC Switch

When not under VCPI, the -HWIVEC switch forces 386 | DOS-Extender to relocate hardware interrupts. If the -HWIVEC switch is used under VCPI, 386 | DOS-Extender relocates hardware interrupts (if the VCPI host has not moved them). If the VCPI host has already relocated hardware interrupts, 386 | DOS-Extender cannot move them again. When the -HWIVEC switch is used, the 250Ch call always reports the actual interrupt vectors for hardware interrupts in protected mode.

For example, suppose the -HWIVEC 78h switch is used. When running without VCPI, or under a VCPI host (such as an EMS emulator) that doesn't relocate hardware interrupts, this causes 386 | DOS-Extender to relocate hardware interrupts IRQ0-7 to vectors 78h-7Fh. The 250Ch call returns 78h for the IRQ0 vector, and the application must install a protected mode INT 78h handler to trap IRQ0 interrupts.

If the -HWIVEC 78h switch is used but DESQview has already remapped IRQ0-7 to vectors 50h-57h, then 386 | DOS-Extender does not relocate hardware interrupts a second time. Instead, the 250Ch call reports vector 50h as the interrupt vector used for IRQ0.

There are several points to keep in mind if you use the -HWIVEC switch:

- Using -HWIVEC defeats the 386|DOS-Extender default operation of making it appear as though hardware interrupts occur on their standard MS-DOS vectors in all environments. The 250Ch call reports the actual hardware interrupt vectors, and the application must install protected mode handlers on those vectors to trap the protected mode hardware interrupts.
- You have to use the 250Ch call to find out which vectors to use; you won't necessarily see IRQ0-7 on the vectors you selected with the -HWIVEC switch. If a VCPI host has already relocated hardware interrupts, you'll see the vectors selected by the VCPI host instead. You may even see the standard MS-DOS vectors, if you're running under DPMI (which doesn't support hardware interrupt relocation).

☛ For real mode interrupt handlers, always install hardware interrupt handlers on the standard MS-DOS interrupt vectors. If the VCPI host relocated hardware interrupts, it always reflects them on the standard MS-DOS vectors in V86 mode, so real mode handlers installed on the relocated vectors won't see the hardware interrupts.

Please see Appendix L for more information on the use of the -HWIVEC switch.

Privilege-Level Zero Operation

The -PRIVILEGED switch forces 386 | DOS-Extender to run the application at privilege level zero (the most privileged level). In version 3.0 of 386 | DOS-Extender, privileged operation is still the default (to ease the transition from earlier versions where level zero was the only possibility); future versions of 386 | DOS-Extender will default to unprivileged execution

Most applications should use the -UNPRIVILEGED switch to run at privilege level three (or a host-selected privilege level under DPMI) for the following reasons:

- Once DPMI is supported in 386|DOS-Extender, applications that run at level zero will not be able to run in a DPMI environment such as Windows 3.0 enhanced mode. This is the most important reason for moving toward unprivileged operation. (Level zero operation is fully compatible with VCPI).
- When running with 386|VMM, the application's stack must be locked if running at privilege level zero. This is not necessarily a serious limitation if the stack is not large and the application doesn't do a lot of stack switching. See the *386|VMM Reference Manual* for more information.
- It's less likely that a bug in the application program will cause a machine reboot with unprivileged operation.

However, for applications that won't be run in a DPMI environment, there can be some good reasons for running at privilege level zero. The fundamental reason for privileged operation is more control over the machine; for example, programs executing at level zero can:

- read and write system registers (debug registers, control registers, test registers)
- use 386|DOS-Extender GDT segments (see Table H-1). In particular, the GDT contains data segments that map the GDT, LDT, and IDT, so level zero applications can directly read and write descriptors in those system tables
- set up task gates to use the task-switching capabilities of the 80386.

TABLE H-1
SELECTED GDT SEGMENTS

Segment Selector	Description
0028h	The LDT system segment. This is the segment selector value loaded into the LDTR register.
0030h	A writable data segment mapping the LDT. This is the segment used by 386 I DOS-Extender to modify segment descriptors in the LDT.
0038h	A writable data segment mapping the GDT. This is the segment used by 386 I DOS-Extender to modify segment descriptors in the GDT.
0050h	A writable data segment mapping the IDT (interrupt descriptor table). This segment is used by 386 I DOS-Extender to modify interrupt descriptors in the IDT.
0060h	A writable data segment mapping the first 1 MB of memory used by MS-DOS. When running under MS-DOS, an offset within this segment is equal to a physical memory address. When running under VCPI, offsets in this segment have no relationship to physical memory address.

Processor Exceptions

When the 80386 processor detects a program error, it generates a processor exception. When an exception occurs, 386 | DOS-Extender prints out a message identifying the exception and the program address at which it occurred, then terminates the program. The program should then be run under the 386 | DEBUG or 386 | SRCBug debuggers, which gain control when the exception occurs and allow you to examine and alter memory and registers, to determine the cause of the exception.

The table below lists the processor exceptions that can occur on the 80386. For more information on exceptions, see Reference 1.

TABLE I-1
PROCESSOR EXCEPTIONS

Exception Name	Interrupt Number	Description
Divide Error	0	Divide by zero with the DIV or IDIV instruction.
Debug Exception	1	This exception is used by debuggers and should never occur at the application level.
Breakpoint	3	This exception is generated by the one-byte INT 3 instruction and is for use by debuggers.
Overflow	4	This exception is generated when the INTO instruction is executed, and the overflow flag is set.

(cont.)

Exception Name	Interrupt Number	Description
Array Bounds Check	5	This exception is generated when the BOUND instruction is executed and the operand exceeds the specified limits.
Invalid Opcode	6	An attempt to execute an illegal instruction opcode. Usually caused by jumping into the middle of data.
Coprocessor Not Available	7	Used for emulating 80287 or 80387 instructions when the coprocessor is not present. This exception occurs when a coprocessor instruction is encountered and the EM bit is set in the processor CR0 register.
Double Fault	8	A second exception occurred during processing of an exception.
Coprocessor Segment Overrun	9	A page or segment violation occurred while transferring a coprocessor instruction operand to the 80287 or 80387 coprocessor. This exception never occurs under 386 | DOS-Extender.
Invalid TSS	10	An invalid task state segment was detected during a task switch. 386 | DOS-Extender does not use task switches.
Segment Not Present	11	The segment descriptor in the LDT or GDT, which is referenced by the segment selector value being loaded into a segment register, is marked not present.
Stack Fault	12	An instruction which uses the SS segment register has exceeded the limit on the stack segment. Usually due to a stack overflow or an invalid segment offset generated with the EBP register.

(cont.)

Exception Name	Interrupt Number	Description
General Protection Fault	13	Can be caused by all sorts of conditions, including segment limit violations, bad segment selector values, and violating the protection settings in a segment descriptor.
Page Fault	14	An attempt to access a page marked not present, or to violate the protection settings in a page table entry. Under 386 I DOS-Extender, usually due to a reference through a null pointer in the program segment.
Coprocessor Error	16	Never generated on PC AT, PS/2, or EISA-compatible hardware. This interrupt is used on MS-DOS systems as a BIOS system call.
Alignment Check (486 only)	17	Never generated under 386 I DOS-Extender. Conflicts with BIOS equipment check system call.

PC Hardware Interrupts

Hardware interrupts on the IBM PC AT, PS/2, and EISA architectures are controlled through two Intel 8259A interrupt controller chips. These devices are programmable, so the interrupt numbers that the hardware interrupts are vectored through can be modified. The table below gives the default hardware interrupt vector locations for 386 | DOS-Extender programs executing under MS-DOS. See section 6.6.1 and the description of 386 | DOS-Extender system function 250Ch for more details.

TABLE J-1.
HARDWARE INTERRUPTS

Hardware Interrupt	Interrupt Number	Description
IRQ0	08h	Timer tick—occurs 18.2 times per second
IRQ1	09h	Keyboard interrupt
IRQ2	0Ah	Not generated—used to connect slave 8259A interrupt controller
IRQ3	0Bh	Serial communications port two
IRQ4	0Ch	Serial communications port one
IRQ5	0Dh	Parallel port two

(cont.)

Hardware Interrupt	Interrupt Number	Description
IRQ6	0Eh	Floppy disk drive controller
IRQ7	0Fh	Parallel port one
IRQ8	70h	Real-time clock—when enabled, occurs once every 976 microseconds
IRQ9	71h	Connected to expansion bus pin B04
IRQ10	72h	Connected to expansion bus pin D03
IRQ11	73h	Connected to expansion bus pin D04
IRQ12	74h	Connected to expansion bus pin D05 on IBM PC AT architectures, the mouse interrupt on PS/2 architectures.
IRQ13	75h	80287 or 80387 coprocessor error (this is generated instead of processor exception 10h)
IRQ14	76h	Fixed disk drive controller
IRQ15	77h	Connected to expansion bus pin D06

Re-entrancy in System Calls

This appendix documents 386 | DOS-Extender operations that are not re-entrant. Applications that perform scheduling and context switching between different tasks must prevent two separate tasks from using non-re-entrant functions simultaneously.

The 386 | DOS-Extender mode switching and interrupt processing is completely re-entrant. Only some of the system calls and the page fault processing performed by 386 | VMM are non-re-entrant because they use global variables. The sections below identify all non-re-entrant operations under 386 | DOS-Extender.

K.1 DOS and BIOS Calls that Use Data Buffering

A single global data buffer (allocated under the control of the -MINIBUF and -MAXIBUF switches) is used to buffer data on DOS and BIOS system calls. If the data is less than 256 bytes in length, 386 | DOS-Extender always buffers the data on the interrupt stack, so there is no re-entrancy problem.

Task switchers must use one of the following techniques to avoid re-entrancy problems with the global data buffer:

- Prevent a task switch while a system call that uses the data buffer is active.
- Save/restore the buffer contents (its address and size are obtainable with system call 2517h) on every context switch.
- Use the -DATATHRESHOLD 0 switch to force 386|DOS-Extender to attempt to convert protected mode pointers to real mode pointers before buffering data, and make sure the application buffers for all non-reentrant system calls are in conventional memory.

The following system calls use the global data buffer for data sizes over 256 bytes. No more than one of these calls can be active at any one time.

INT	Function	Name
21h	09h	Output Character String
21h	3Fh	Read File
21h	40h	Write File
21h	44h	Device I/O Control
21h	56h	Rename File
21h	65h	Get Extended Country Information
10h	10h	Color Control
10h	13h	Write Character String
10h	1Ch	Save/Restore Video State

K.2 Memory Allocation Calls and Page Fault Processing

The memory management subsystem of 386 | DOS-Extender uses a number of global variables. Only one of the memory management system calls below can be active at any one time. The page fault handler in 386 | VMM also uses memory management globals. The 2520h system call can be used to determine whether a 386 | VMM page fault is currently active.

INT	Function	Name
21h	48h	Allocate Segment
21h	49h	Free Segment
21h	4Ah	Resize Segment
21h	250Ah	Map Physical Memory at End of Segment
21h	2512h	Load Program for Debugging
21h	2516h	Free All Memory Owned by LDT
21h	2517h	Get Info on DOS Data Buffer
21h	2519h	Get Additional Memory Error Information
21h	251Ah	Lock Pages in Memory
21h	251Bh	Unlock Pages
21h	251Ch	Free Physical Memory Pages
21h	2520h	Get Memory Statistics
21h	2521h	Limit Program's Extended Memory Usage
21h	2525h	Limit Program's Conventional Memory Usage
21h	2529h	Load Flat Model .EXP or .REX File
21h	252Ah	Load Program for Debugging

INT	Function	Name
21h	252Bh	Memory Region Page Management
21h	252Ch	Add Unmapped Pages at End of Segment
21h	2530h	Set DOS Data Buffer Size
21h	2536h	Minimize/Maximize Extended/Conventional Memory Usage

K.3 Miscellaneous Non-Re-entrant System Calls

The following system calls all use the same global buffer; only one of them can be active:

INT	Function	Name
21h	2512h	Load Program for Debugging
21h	2529h	Load Flat Model .EXP or .REX File
21h	252Ah	Load Program for Debugging
21h	2530h	Set DOS Data Buffer Size
21h	2539h	Get .EXP Header Offset in Bound File

The following system calls use an internal global DTA buffer in 386 | DOS-Extender; only one of them can be active:

INT	Function	Name
21h	4Eh	Search for First Match
21h	4Fh	Search for Next Match

The 2501h system call, Reset 386 | DOS-Extender Data Structures, removes the context of any active mode switches.

Compatibility with Earlier Versions of 386 | DOS-Extender

This appendix documents the major differences between version 3.0 and earlier versions of 386 | DOS-Extender.

L.1 Hardware Interrupt Relocation (-HWIVEC)

Version 3.0 of 386 | DOS-Extender does not relocate hardware interrupts IRQ0-7 from the default MS-DOS interrupt vectors of 08h-0Fh. The -HWIVEC switch can be used (but is not recommended) to force 386 | DOS-Extender to relocate IRQ0-7.

Earlier versions of 386 | DOS-Extender relocated IRQ0-7 to interrupt vectors 78h-7Fh by default. The -INTMAP 8 switch can be used to defeat this hardware interrupt relocation.

This change means that interrupts 08h-0Fh are used for both hardware interrupts and processor exceptions in version 3.0, so the application uses different system calls (2532h and 2533h) to get and set protected mode processor exception handlers in 08h-0Fh range. If the -INTMAP 8 switch is used with versions of 386 | DOS-Extender prior to 3.0, processor exceptions in the 08h-0Fh range cannot be intercepted.

A second change appears in the real mode interrupt processing performed by 386 | DOS-Extender if hardware interrupts **are** relocated (the -HWIVEC switch is used under version 3.0). In version 3.0, handlers are installed at the relocated vectors, and they chain to the handler for the vector at the standard MS-DOS location. For example, if the -HWIVEC 78h switch is used, 386 | DOS-Extender installs a real mode INT 78h handler that picks up the vector for INT 08h and jumps to that handler. Applications that install real mode hardware interrupt handlers should use the standard MS-DOS vectors even if hardware interrupt are relocated. It is not possible to install real mode handlers for processor exceptions 08h-0Fh even if -HWIVEC is used.

By contrast, versions earlier than 3.0 take the real mode vectors at the standard MS-DOS locations and copy them to the relocated hardware interrupt vectors. Applications that install real mode hardware interrupt handlers must use the relocated interrupt vectors, and interrupts 08h-0Fh are used only for processor exceptions (not for hardware interrupts).

L.2 Task Switching on Hardware Interrupts

Versions of 386 | DOS-Extender starting with 2.2 and prior to 3.0 perform task switching on hardware interrupts to get a new stack when a hardware interrupt occurrs, so the application stack can be virtualized under 386 | VMM. In version 3.0, the same thing is accomplished by running applications at privilege level three; level zero applications must run with a locked stack under 386 | VMM.

Hardware interrupts invoked with a task switch have nothing (not even EFLAGS and CS:EIP) on their initial stack frame. Instead, they must look in the back-linked TSS to see the register values at the time of the interrupt. 386 | DOS-Extender uses the convention that every system TSS segment in the GDT is immediately followed by a data segment mapping that TSS. Therefore, a hardware interrupt handler running under version 2.x (x >= 2) of 386 | DOS-Extender can obtain addressability of the back-linked TSS with the following code:

```
    ltr     ax                  ; get data seg for current TSS
    add     ax,8                        ;
    mov     es,ax                       ;
    mov     ax,es:0             ; get back-linked TSS
    add     ax,8                ; get data seg for back-linked TSS
    mov     es,ax                       ;
```

An important property of TSSs is that they cannot be reentered. The 80386 processor generates a general protection exception on an attempt to enter a TSS that is already marked busy. This means that protected mode hardware interrupt handlers under versions 2.x (x >= 2) of 386 | DOS-Extender must take steps to ensure they cannot be reentered. This is done by making sure that interrupts are disabled from the time the EOI is sent to the 8259 until the IRETD that terminates the handler.

L.3 Interrupt Stack Frames

The interrupt stack frames documented in Chapter 6 were added in version 3.0, as part of the support for privilege level three operation. Earlier versions of 386 | DOS-Extender vectored directly to interrupt handlers through the LDT. The result was that the interrupt handler got control on whatever stack was active when the interrupt occurred, with the EFLAGS and CS:EIP from the interrupt pushed on the stack.

L.4 MS-DOS Sharing Mode on .EXP File Open

386 | DOS-Extender performs a DOS Open File (3Dh) system call when loading the application .EXP file, and when loading a program with the 2512h, 2529h, and 252Ah system calls. If 386 | VMM is present and the file is unpacked (the -PACK linker switch was not used), the file is kept open until program termination so 386 | VMM can page out of the .EXP file to save space in the swap file.

Version 3.0 of 386 | DOS-Extender opens the .EXP file read only, no child inheritance, with MS-DOS sharing mode set to compatibility mode (AL = 80h). Previous versions of 386 | DOS-Extender opened the file read only, no child inheritance, with deny write compatibility mode (AL = A0h).

This change was made because when MS-DOS loads a program, it opens the file read-only in compatibility mode. For bound .EXE files (where both the 386 | DOS-Extender and the application are in the same file), the file is opened first by MS-DOS to load 386 | DOS-Extender, and then by 386 | DOS-Extender to load the application. If 386 | VMM is used, the file is kept open until the application program terminates.

This causes no problems when the program is run only once. However, there is a potential problem if the program is run from a network server by many users simultaneously. In this case, if 386 | VMM is used, the file is simultaneously kept open by many users, so it's important that all the sharing modes used are permitted for multiple opens. Specifically, suppose user A is running a bound program with 386 | VMM; then the .EXE file is kept open in the sharing mode used for 386 | DOS-Extender open file call. When user B runs the same program, MS-DOS opens the .EXE file in compatibility mode.

The strict interpretation of sharing modes dictates that a simultaneous read only write deny, and read only compatibility mode, open is permitted only if the file itself is marked read only. In fact, most networks permit this combination even if the file is not read only; but Microsoft's LAN Manager and SHARE.EXE products, at least, apply the strict interpretation. To avoid problems with bound applications in those environments, the sharing mode used by 386 | DOS-Extender was changed to compatibility mode in the 3.0 release.

The workaround for problems with file sharing in earlier versions of 386 | DOS-Extender is to mark the bound .EXE file read only with the MS-DOS ATTRIB command:

```
   attrib filename +r
```

L.5 Re-entrancy

Versions of 386 | DOS-Extender prior to 3.0 had more re-entrancy restrictions than those documented in Appendix K. All system calls that took a pointer as input used the global data buffer; none of them buffered data on the stack. In addition, all active mode switches had to be nested; it was not possible to suspend a task with active mode switches and return to it later.

Glossary

386	The Intel 80386 microprocessor.
386SX	The Intel 80386SX microprocessor, identical to the 80386 from a software point of view.
386 | VMM	The Phar Lap Virtual Memory Manager.
486	Intel's 80486 microprocessor, from a software point of view just a fast 386 with a built-in numeric coprocessor.
A20	Address line 20. When enabled, allows full 32-bit addressing; when disabled, truncates addresses to 20 bits. Normally disabled under MS-DOS real mode to force the address space wrap-around occurring at one megabyte on 8088 and 8086 systems. Always enabled when running a program in protected mode under 386 | DOS-Extender.
aliased segments	Segments that share the same memory locations. The application program code and data segments are aliased. If one of a set of aliased segments is resized, all of the aliased segments are automatically resized.
allocated pages	Memory pages that contain application program code and data. Allocated pages are created when an application program is loaded, and when it allocates a segment or increases the size of a segment.

command file	Optional file containing command line switches that 386 I DOS-Extender will use in executing the application if the file is present .
command line switches	Switches on a program's invocation line specifying aspects of its environment or desired processing. A minus sign (-) always precedes the switch name.
COMPAQ built-in memory	256KB of RAM memory, allocated through a Compaq-specific interface on COMPAQ 386 and 486 PCs, hardware mapped to just below 16 MB.
conventional memory	Memory below 1 MB, obtained from MS-DOS.
Cyrix coprocessor	The Cyrix EMC87 floating point coprocessor, which can be programmed either with standard 80387 instructions, or through a memory-mapped interface that affords higher performance. The Cyrix coprocessor can be programmed through its memory-mapped interface by using segment 004Ch under 386 I DOS-Extender.
DESQview 386	A multitasking environment from Quarterdeck Office Systems. 386 I DOS-Extender is fully compatible with DESQview.
direct extended memory	Memory above 1 MB not used by any other program and not available through XMS or VCPI. 386 I DOS-Extender allocates this memory directly; please see section 5.2.2.
DOSX	The environment variable which 386 I DOS-Extender always reads for command line switches.

DPMI	DOS Protected-Mode Interface. An interface in Windows 3.0 enhanced mode, likely to be supported later in OS/2, UNIX, and other multitasking environments.
EISA	Extended Industry Standard Architecture, a PC bus architecture introduced in 1989.
EMS	The Lotus-Intel-Microsoft [LIM] Expanded Memory Specification. On 386 and 486 machines, EMS memory is usually provided by EMS emulators such as the Quarterdeck QEMM-386 and the Qualitas 386-to-the-Max.
environment strings	Strings maintained in the MS-DOS environment buffer, settable and retrievable by users or applications. Segment 002Ch always points to the application's environment buffer.
exception	An interrupt generated by the 80386 processor when it detects a software error.
expanded memory	Memory allocated through the EMS interface.
extended memory	Memory above 1 MB, from all sources other than MS-DOS conventional memory: VCPI/EMS, XMS, COMPAQ built-in, or direct extended.
GB	Gigabyte, one billion bytes.
GDT	Global Descriptor Table, one of two system tables (with the LDT) controlling protected mode segments.
hardware interrupt	An asynchronous interrupt generated by external hardware. There are sixteen possible hardware interrupts on most 386/486 PCs.

hexadecimal	Numbers based on 16 rather than 10, designated with a suffix of "h". For example, 100h = 256 decimal.
IDT	Protected mode Interrupt Descriptor Table, used when a hardware interrupt, software interrupt, or processor exception occurs in protected mode to vector to the correct interrupt handler.
ISA	Industry Standard Architecture, the PC bus architecture used in the original IBM PC AT.
IVT	Real mode Interrupt Vector Table, used when a hardware interrupt, software interrupt, or processor exception occurs in real mode to vector to the correct interrupt handler.
LDT	Local Descriptor Table, one of two system tables (with the GDT) controlling protected mode segments.
linear address	Memory addresses seen by an application program. The linear address of a piece of code or data is equal to the base address of the segment plus the offset of the code or data within the segment. Linear addresses are translated to physical addresses by the page tables, which are system tables managed by 386 | DOS-Extender.
locked memory	Under VMM, pages kept in physical memory and not swapped out to disk.
MB	Megabyte, one million bytes.
MCA	Micro Channel Architecture, the PC bus architecture used in IBM PS/2 machines.
memory pages	4 KB units of memory, the standard unit of memory allocation in 386 protected mode.

memory paragraphs	16-byte units of memory, the standard unit of memory allocation under MS-DOS.
page tables	A table within a computer that contains a mapping between linear page addresses and physical page addresses.
physical address	The memory address placed on the address bus by the 386/486 processor. Under 386 I DOS-Extender, application programs see linear addresses, not physical addresses.
physical device pages	Memory pages used to access a memory-mapped hardware device, such as the Weitek coprocessor, the Cyrix coprocessor, or video memory. Physical device pages do not consume any physical RAM memory or 386 I VMM swap file resources.
physical memory	Actual physical RAM memory (as opposed to virtual memory).
privilege level	One of four privilege levels (also called rings) supported by the 386/486, zero being the most privileged and three the least privileged. Under 386 I DOS-Extender, application programs can run at level zero or level three.
protected mode	The native, 32-bit operating mode of the 386/486. This is the operating mode used for application programs under 386 I DOS-Extender.
PSP	Program Segment Prefix, a data structure allocated by MS-DOS for each active program.
real mode	The 8086-compatible operating mode of the 386/486. This is the operating mode used by MS-DOS, which is an 8086 operating system.

ring	Please see privilege level.
segment	In real mode, a sequential area up to 64 KB in length, referenced by a 16-bit offset. In protected mode, up to four GB long, using a 32-bit offset.
segment descriptor	An 8-byte entry in the GDT or LDT that defines a protected mode segment. It includes the segment base, segment limit, and access rights.
selector	The value loaded into a segment register. In real mode or V86 mode, this is the memory paragraph address of the segment. In protected mode, a selector references a segment descriptor in the GDT or LDT.
software interrupt	An interrupt generated by a software INT instruction. Software interrupts are used for DOS, BIOS, and 386 I DOS-Extender system calls.
super extended memory	Memory above 16 MB on EISA machines. Super extended memory is allocated by 386 I DOS-Extender through the XMS interface.
unmapped pages	Memory pages marked not present, and which cause a page fault when referenced by the application program. Unmapped pages are created with the -OFFSET switch or a 386 I DOS-Extender system call.
V86 mode	An 8086-compatible operating mode of the 386/486, which can only be provided under the control of a protected mode system program. V86 mode is used by EMS emulators and multitasking environments such as Windows 3.0, DESQview 386, and OS/2. 386 I DOS-Extender can only run in a V86 mode environment if the VCPI or DPMI interface is present.

VCPI	Virtual Control Program Interface, provided by most 386 EMS emulators, such as the Microsoft EMM386, the Quarterdeck QEMM-386, and the Qualitas 386-to-the Max. VCPI allows 386 | DOS-Extender to be compatible with EMS emulators, and also with the Quarterdeck DESQview 386 multitasker.
virtual memory	Under 386 | VMM, memory allocated to the application is virtual memory, which means it may or may not actually be in physical memory. Virtual pages not in physical memory are kept swapped out to disk. Virtual memory allows an application to run on a PC with less physical memory than the amount needed by the program.
Weitek coprocessor	The Weitek 1167, 3167, or 4167 floating point coprocessor. This high–performance coprocessor uses a memory-mapped interface, and can be programmed using segment selector 003Ch under 386 | DOS-Extender.
Windows	A multitasking environment from Microsoft. 386 | DOS-Extender is compatible with the real and standard mode of Windows. Compatibility with Windows enhanced mode requires DPMI support, and is due in the next major 386 | DOS-Extender release after 3.0
XMS memory	Memory allocated through the XMS (eXtended Memory Specification) interface, a Microsoft standard for allocating extended memory. An XMS driver is usually called HIMEM.SYS.

Error Messages

Errors in 386 | DOS-Extender are divided into two categories:

- errors, which occur during initialization, before the application program begins to run (e.g., insufficient memory to load the program)
- fatal errors, which occur while the application is executing, when a condition occurs from which 386|DOS-Extender cannot recover.

For either category of error, 386 | DOS-Extender prints an error message with a unique error number and a brief description of the cause of the error.

386 | DOS-Extender further classifies errors into:

- user errors, or errors that are probably caused by either a bug in the application program or a program requirement that is not satisfied on that machine
- system errors, or errors that are probably the result of a bug in 386|DOS-Extender.

This appendix contains a description of all user errors that can occur.

In the unlikely event that a system error occurs, you should call Phar Lap technical support for assistance (617–661–1510). Please be prepared to provide the following information:

1. the error number and the exact text of the error message
2. the manufacturer and model number of the PC, how much memory it has, and if possible the BIOS manufacturer

3. the contents of the CONFIG.SYS and AUTOEXEC.BAT files on the machine
4. whether the condition can be re-created, and if so, the steps that must be taken for the error to occur.

A user error is identified by the following error message format:

```
Phar Lap err n:  error message
```

A user fatal error is identified by the following message format:

```
Phar Lap fatal err n:  error message
```

System errors are identified by the following message formats:

```
Phar Lap sys err n:  error message
Phar Lap fatal sys err n:  error message
```

N.1 User Errors

```
Phar Lap err 1 Invalid baud rate specified with -BAUD switch.
```

Description: Valid baud rates are 110, 150, 300, 600, 1200, 2400, 4800, or 9600. If you need a different baud rate, use the DOS MODE command to set the port parameters and don't use the -BAUD switch.

```
Phar Lap err 2 Channel initialization for I/O redirection
failed.
```

Description: Check to make sure you're using the correct serial port. Use -COM1 for port 1 and -COM2 for port 2. Try using the DOS MODE command to set the port parameters and not using the -BAUD switch.

```
Phar Lap err 3 Serial channel does not exist – I/O redirection
failed.
```

Description: Use -COM1 for serial port 1, -COM2 for serial port 2.

```
Phar Lap err 4 No .EXP file specified.
```

Description: Enter the name of your .EXP file on the command line.

```
Phar Lap err 5 System does not have an 80386 or later
processor.
```

Description: Do not run 386 | DOS-Extender on 8086- or 286-based PCs.

```
Phar Lap err 6 QEMM must be on to run under DESQview.
```

Description: Install QEMM-386 in your CONFIG.SYS file.

```
Phar Lap err 7 MS-DOS version 3.0 or later required.
```

Description: Do not run 386 | DOS-Extender under versions of MS-DOS prior to 3.0.

```
Phar Lap err 8 Can't run in OS/2 compatibility box.
```

Description: 386 | DOS-Extender is not currently compatible with OS/2.

```
Phar Lap err 9
```

Description: The 80386 chip on this system does not always obtain correct results for 32-bit multiply instructions. If you are sure your program does not use any 32-bit MUL instructions, you can force the program to run with the -NOMUL command line switch.

Some Intel 80386 chips have a chip errata that can cause erroneous results from a 32-bit MUL instruction. Either get a new chip, or make sure your program doesn't use the MUL instruction.

```
Phar Lap err 10
```

Description: The 80386 chip in your system was manufactured prior to step B1 of the chip. Chip steps prior to B0 have a bug that makes them unable to switch from protected mode back to real mode. If you are sure that the 80386 chip in your system is step B0, then use the -B0 switch on the command line to force this program to run. If you use the -B0 switch with a chip earlier than step B0, it will crash your system!

386 | DOS-Extender can run on B0 stepping 80386 chips, but not A steppings. If you know your computer has a B0 chip step, use the -B0 switch to tell 386 | DOS-Extender it's OK to run.

```
Phar Lap err 11 Couldn't obtain HW interrupt vectors from EMS
emulator.
```

Description: The EMS emulator in your CONFIG.SYS file doesn't correctly implement the VCPI interface. Remove it or use QEMM-386 or 386-to-the-MAX instead.

```
Phar Lap err 12 Bad value for -HWIVEC or -INTMAP switch.
```

Description: The -HWIVEC and -INTMAP switches must be a multiple of 8, must be between 30h and F8h, and must not conflict with the -PRIVEC switch (default value 80h).

```
Phar Lap err 13 Illegal switch values:  -MINIBUF n, -MAXIBUF n.
Illegal switch values:  -EXTLOW n, -EXTHIGH n.
```

Description: The -MINIBUF and -MAXIBUF switches must be between 1 and 64, and -MAXIBUF must be >= -MINIBUF. The -EXTLOW switch must be >= one megabyte (100000h).

```
Phar Lap err 14  -NISTACK must be between 6 and n
```

Description: The -NISTACK switch cannot be less than 6 or greater than the upper bound specified in the error message.

```
Phar Lap err 15 A maximum of 63 KB may be alloc'd with -NISTACK
and -ISTKSIZE.
```

Description: The product of the -NISTACK and -ISTKSIZE switch values cannot be greater than 63.

```
Phar Lap err 16 -CALLBUFS must be < = 64 KB.
```

Description: The -CALLBUFS switch value cannot be greater than 64.

```
Phar Lap err 18 -NOPAGE cannot be used with -OFFSET or
-REALBREAK.
```

Description: Stop using conflicting switches.

```
Phar Lap err 19 -MINSWFSIZE cannot be greater than -MAXSWFSIZE.
```

Description: Adjust the values appropriately.

```
Phar Lap err 20 -VSCAN value must be between n and m.
```

Description: Adjust the value as specified.

```
Phar Lap err 21 Can't use 386|VMM with -NOPAGE or real-mode pgm.
```

Description: Stop using -NOPAGE. You may want to use -ERRATA17 with 386|VMM if your application uses the 387 coprocessor.

```
Phar Lap err 22  -SWAPCHK ON used, but unable to get info on
swap device:  X.
```

Description: The MS-DOS Get Disk Space call (36h) returns an error for the device on which the swap file is located. If it's a network device, check to see if the network software has a bug. 386|VMM must be able to get the free disk space in order to run with the -SWAPCHK ON switch setting.

```
Phar Lap err 23 EMS emulator returned error on 'get physical
address' call.
```

Description: The EMS emulator in your CONFIG.SYS file doesn't correctly implement the VCPI interface. Remove it or use QEMM-386 or 386-to-the-MAX instead.

`Phar Lap err 24 Can't alloc system conventional mem buffer for GDT and LDT.`

Description: There is insufficient conventional memory available for the needs of 386|DOS-Extender. A memory requirements summary is also printed showing how much memory is needed. Use the -DEBUG 3 switch to get more detailed memory allocation data.

`Phar Lap err 25 Relative paths not permitted with -SWAPDIR.`

Description: Specify an absolute path.

`Phar Lap err 29 Bad value for -PRIVEC switch.`

Description: The -PRIVEC switch value must be between 30h and FFh, and must not conflict with -HWIVEC, -INTMAP, or vectors used for hardware interrupts (normally 70h-77h).

`Phar Lap err 30 Can't use -A20 with XMS driver or VCPI (EMS emulator)`

Description: Stop using the -A20 switch.

`Phar Lap err 31 Reg CR0 has bad value for real mode:` *`value`*

Description: Call Phar Lap technical support at (617) 661-1510.

`Phar Lap err 33 Can't enable address line 20.`

Description: There's a hardware problem with your PC. Call Phar Lap technical support at (617) 661-1510.

```
Phar Lap err 34  Unable to turn off program using virtual 8086
mode.
```

Description: Turn it off manually or take it out of CONFIG.SYS.

If the program using V86 mode is an EMS emulator that doesn't support VCPI, replace it with QEMM-386 or 386-to-the-MAX.

```
Phar Lap err 35
```

Description: The 386 chip is currently executing in virtual 8086 mode under the control of another program. You must turn off this other program in order to use 386 | DOS-Extender to run in protected mode.

If the program using V86 mode is an EMS emulator that doesn't support VCPI, replace it with QEMM-386 or 386-to-the-MAX.

```
Phar Lap err 36 EMS emulator reports VCPI but doesn't use V86
mode.
```

Description: The EMS emulator in your CONFIG.SYS file doesn't correctly implement the VCPI interface. Remove it or use QEMM-386 or 386-to-the-MAX instead.

```
Phar Lap err 37 Cannot use -NOPAGE under V86 mode environment.
```

Description: Stop using -NOPAGE, or remove the EMS emulator or other program that is running in V86 mode.

`Phar Lap err 38 You must use version 2.2 or later of BIND386 to bind your program.`

Description: Use the copy of BIND386 that came with your bindable 386 | DOS-Extender.

`Phar Lap err 39 Bindable 386|VMM driver can only run if bound with BIND386.`

Description: Use the copy of BIND386 that came with your bindable 386 | DOS-Extender.

`Phar Lap err 40 Development 386|VMM driver cannot be bound with BIND386.`

Description: Use the -VMFILE switch to load the development version of 386 | VMM.

`Phar Lap err 41 Not a Phar Lap 386|VMM file:` *`filename`*`.`

Description: The file selected with the -VMFILE switch has been corrupted, or is not 386 | VMM from Phar Lap Software.

`Phar Lap err 42 Incompatible version of 386|VMM file.`

Description: Use the same version number of 386 | VMM as 386 | DOS-Extender. You can obtain the version numbers from the distribution disks, or by using the -BANNER switch.

`Phar Lap err 43.Invalid configuration block in 386|VMM file.`

Description: The file selected with the -VMFILE switch has been corrupted, or is not 386 | VMM from Phar Lap Software.

```
Phar Lap err 44 Couldn't init page table from EMS emulator.
```

Description: The EMS emulator in your CONFIG.SYS file doesn't correctly implement the VCPI interface. Remove it or use QEMM-386 or 386-to-the-MAX instead.

```
Phar Lap err 45 Can't open .EXP file:  filename.
```

Description: Check the spelling of the filename, and check to see if it is in a directory on the environment PATH.

```
Phar Lap err 46 Couldn't get free mem size from EMS emulator.
```

Description: The EMS emulator in your CONFIG.SYS file has a bug. Remove it or use QEMM-386 or 386-to-the-MAX.

```
Phar Lap err 47 Can't read in header for .EXP file:  filename.
```

Description: The file has been corrupted, or is not an .EXP file.

```
Phar Lap err 48 -GDTENT or -LDTENT value too large.
```

Description: The -GDTENT and ~LDTENT switch values must be <= 8192.

```
Phar Lap err 49 Can't allocate conventional memory for 386|VMM.
```

Description: There is insufficient conventional memory available for the needs of 386|DOS-Extender. A memory requirements summary is also printed showing how much memory is needed. Use the -DEBUG 3 switch to get more detailed memory allocation data.

`Phar Lap err 55 Can't open 386|VMM file:` *`filename`*`.`

Description: Check the spelling of the file name, and whether it is in a directory on the the environment PATH.

`Phar Lap err 56  Can't read 386|VMM file:` *`filename`*`.`

Description: The file is corrupted.

`Phar Lap err 58 Can't create VM swap file of size` *`n`* `in directory` *`dirname`*`.`

Description: There is insufficient swap space on the selected swap device. Use the -SWAPDIR switch to select a different swap device.

`Phar Lap err 70 Can't create -PAGELOG file:` *`filename`*`.`

Description: A DOS error occurred attempting to create the file specified with the -PAGELOG switch. Check the spelling of the file name and path.

`Phar Lap err 72 -VSLEN value must be >=` *`n`*`.`

Description: The -VSLEN switch value must be larger than the number shown in the error message.

`Phar Lap err 73 The -HWIVEC and -INTMAP switches can't both be used.`

Description: Remove one of the switches.

```
Phar Lap err 10013 Insufficient conventional memory for data
buffers.
```

Description: There is insufficient conventional memory available for the needs of 386|DOS-Extender. A memory requirements summary is also printed showing how much memory is needed. Use the -DEBUG 3 switch to get more detailed memory allocation data.

`Phar Lap err 10028 Not a protected-mode program:` *`filename`*`.`

Description: Not an .EXP file, or the file has been corrupted.

N.2 User Fatal Errors

```
Phar Lap fatal err 10030 386|VMM:  Couldn't reopen 386|VMM swap
file after flushing it.
```

Description: On a multitasking system, too many file handles are in use at once. Reduce the number of simultaneously running programs, or don't use the -FLUSHSWAP switch to flush the swap file, and use the 25C3h form of the EXEC call rather than 4Bh to avoid flushing the swap file on an EXEC.

```
Phar Lap fatal err 10031 386|VMM:  Page fault handler was
reentered.
```

Description: A hardware interrupt handler or a critical error handler caused a page fault, or you are running at privilege level zero (-UNPRIVILEGED) and the stack memory is not locked. Make sure all memory used by hardware interrupt handlers (code, data, and stack) is locked.

```
Phar Lap fatal err 10032 386|VMM:  Bad page table entry in pg
flt hndlr 1st err val = linadr, 2nd = pg tbl entry, 3rd = pg
tbl info.
```

Description: If you install your own page replacement handlers, they have a bug. Otherwise, call Phar Lap technical support at (617) 661-1510.

```
Phar Lap fatal err 10033 386|VMM:  Page replacement routine
selected an invalid page:  1st err val = linadr, 2nd = pg tbl
entry, 3rd = pg tbl info.
```

Description: If you install your own page replacement handlers, they have a bug. Otherwise, call Phar Lap technical support at (617) 661-1510.

```
Phar Lap fatal err 10034 386|VMM:  Error reading .EXP file, or
data file mapped in virtual memory.
```

Description: Check to see if the file got renamed or deleted. The user may have deleted the file if your program has the capability to run a DOS shell to let the user run any DOS command.

```
Phar Lap fatal err 10035 386|VMM:  Error reading swap file.
```

Description: Check to see if the file got renamed or deleted. The user may have deleted the file if your program has the capability to run a DOS shell to let the user run any DOS command.

```
Phar Lap fatal err 10036 386|VMM:  Page table entry not present
in pg fault hndlr.
```

Description: If you install your own page replacement handlers, they have a bug. Otherwise, call Phar Lap technical support at (617) 661-1510.

```
Phar Lap fatal err 10037 386|VMM:  Error writing swap file.
```

Description: Check to see if the file got renamed or deleted. The user may have deleted the file if your program has the capability to run a DOS shell to let the user run any DOS command.

```
Phar Lap fatal err 10038 386|VMM:  Out of space for swap file.
```

Description: There is no more disk space available on the swap device. Use the -SWAPDIR switch to select a different swap device,. or use -SWAPCHK FORCE to make sure this error cannot happen, or write an out-of-swap-space handler in your program.

```
Phar Lap fatal err 10049 Ran out of stack buffers.
```

Description: Use the -NISTACK switch to increase the number of allocated stack buffers, or don't nest mode switches more than three levels deep.

```
Phar Lap fatal err 10050 Attempted block move (INT 15h func
87h) to memory owned by 386|DOS-Extender:  1st error value =
block base, 2nd error value = word len of block.
```

Description: A program (such as a disk cache driver or a RAM disk driver) installed in your CONFIG.SYS or AUTOEXEC.BAT files is attempting to modify extended memory it does not own. Remove the buggy program.

```
Phar Lap fatal err 10051 Ran out of machine state structs.
```

Description: You are nesting mode switches too many levels deep. Re-architect your program to do less nested mode switching.

```
Phar Lap fatal err 10052 Couldn't reopen 386|VMM page log file
after flushing it.
```

Description: On a multitasking system, too many file handles are in use at once. Reduce the number of simultaneously running programs, or don't use the -FLUSHSWAP switch to flush the page log file, and use the 25C3h form of the EXEC call rather than 4Bh to avoid flushing the page log file on an EXEC.

```
Phar Lap fatal err 10057 Data too large for buffer for INT 21h,
func num shown as err value.
```

Description: Use a higher value for the -MINIBUF switch to allocate a larger data buffer for DOS system calls.

Index

The people who wrote, edited, revised, reviewed, indexed, formatted, polished, and printed this manual were:

Alan Convis, Noel Doherty, Lorraine Doyle, Bryant Durrell, Diego Escobar, Nan Fritz, Jeff Levinger, Elliot Linzer, Bob Moote, Kim Norgren, Amy Weiss, Hal Wadleigh, and Rick Wesson.

60 Aberdeen Avenue, Cambridge, MA 02138
617-661-1510 Fax: 617-876-2972